CHAPTER 6: TWO PROCESSES

6-1. Introduction

6-1.1. Arrangements and Configurations

In a 2x2 situation, when there are two processes to be assigned to one or two identical processors, there are only two possible arrangements: both processes on one processor, or one process on each of two processors. We denote the first arrangement by 2:0 and the second by 1:1.

There is only one configuration in the 2:0 arrangement, 01/-, that is, processes 0 and 1 are on one processing node and there are no processes on the other node. Similarly, there is only one configuration in the 1:1 arrangement, 0/1, that is, process 0 on one processing node and process 1 on the other. We use the labels *APART* for the 0/1 configuration, and *TOGETHER* for the 01/- configuration. Since we are considering only identical processors, it does not matter which processor is used and which unused in the 01/- configuration, nor does it matter which process is on which processor in the 0/1 configuration.

6-2. The Experiment Sets and the Winning Configurations

In this chapter we explore the 2x2 situation. In this section we describe the design and execution of the experiments, and in the following sections we analyze their outcomes.

6-2.1. Experiment Set N

The first set of experiments involve programs made up of two processes, and is based on the full grid (see Section 4-2). We call the two-process full grid experiment set *experiment set N*. Each case in set N is named as shown in Figure 6-1.

	IO-COMM									
CPU	IO 0	COMM 100	IO 25	COMM 75	IO 50	COMM 50	IO 75	COMM 25	IO 100	COMM 0
0	**na**		**nb**		**nc**		**nd**		**ne**	
20	**nf**		**ng**		**nh**		**ni**		**nj**	
40	**nk**		**nl**		**nm**		**nn**		**no**	
60	**np**		**nq**		**nr**		**ns**		**nt**	
80	**nu**		**nv**		**nw**		**nx**		**ny**	
100	**nz**									

Figure 6-1:
Case Labels in Experiment Set N

Each of the 26 cases in set N is repeated twice, that is, two traces are generated for each case, making a total of 52 *runs* in experiment set N. Each trace has approximately 400 actions per process, for a total of approximately 800 actions per trace in set N. (See Section 5-7 for further discussion of trace lengths). Each trace is simulated in each of the two possible configurations. There is no outside load on any part of the simulated computer systems.

The response time of the program, executed in a particular configuration, is the completion time of the last-finishing process. The most obvious result of the simulations is whether the program completes faster when run in the APART configuration or in the TOGETHER configuration. If the program's response time in the APART configuration is faster, we say that APART *wins;* if the TOGETHER response time is faster, we say that TOGETHER wins. Figure 6-2 shows which configuration is fastest for each run in experiment set N.

	IO-COMM									
	IO	COMM	IO	COMM	IO	COMM	IO	COMM	IO	COMM
CPU	0	100	25	75	50	50	75	25	100	0
0	**na**	T T	**nb**	A A	**nc**	A A	**nd**	A A	**ne**	A A
20	**nf**	A A	**ng**	A A	**nh**	A A	**ni**	A A	**nj**	A A
40	**nk**	A A	**nl**	A A	**nm**	A A	**nn**	A A	**no**	A A
60	**np**	A A	**nq**	A A	**nr**	A A	**ns**	A A	**nt**	A A
80	**nu**	A A	**nv**	A A	**nw**	A A	**nx**	A A	**ny**	A A
100	**nz**	A A								

A: APART is the faster configuration
T: TOGETHER is the faster configuration

Figure 6-2:
Winning Configurations in Set N

APART is the faster configuration in every run except the **na** runs, in which TOGETHER was faster. We had expected that the TOGETHER configuration would be faster only in cases where there is a lot of communication, but we had not expected the region where TOGETHER is faster to be restricted to a single grid cell.

There is generally some variation between the grid parameters specified for a cell and the actual proportions of actions in the traces. In the **na** runs there are no execute or I/O actions; all actions are sends or receives except the final terminate action for each process. In cases **nf** and **ng** the actual proportion of execute actions ranges from 14% to 23%. In cases **nb** and **ng** the proportion of IO-COMM which are communication actions ranges from 71% to 77%. Thus, the transition between APART being faster and TOGETHER being faster happens somewhere between 0% and 14% execute actions, and somewhere between 100% and 71% communication, in other words, somewhere within cell **na**.

6-2.2. Experiment Set P

In experiment set N, APART is faster in all runs except **na-1** and **na-2**. Clearly, TOGETHER is faster only in cases where there is a great deal of communication relative to the amounts of CPU and I/O activity. To explore the transition between APART winning and TOGETHER winning, we must concentrate

on the high-communication part of the process-characteristic space. In experiment set P, we ‘‘zoom in’’ on the region represented in set N by case **na**, expanding it to make a new grid, called the *intermediate grid.*

The intermediate grid covers 0-20% CPU in steps of 5%, and 0-25% I/O (100-75% communication) also in steps of 5%, making a 5-by-6 grid of 30 cells. The two-process experiment set based on this grid is called experiment set P; see Figure 6-3.

The four cells in the upper left corner of set N are duplicated in the four outside corner cells of set P. We ignore the duplication; new traces are generated and simulated for the corner cells of set P.

As in set N, two traces are generated for each cell in the grid, making 60 runs. Recall that each run involves two simulations: the APART configuration and the TOGETHER configuration. All other specifications, parameters, and procedures are the same as for set N.

Simulating the traces in experiment set P produced the same winning configurations result as in set N. As shown in Figure 6-3, APART is always faster except in the upper left corner cell.

	IO-COMM											
CPU	IO 0	COMM 100	IO 5	COMM 95	IO 10	COMM 90	IO 15	COMM 85	IO 20	COMM 80	IO 25	COMM 75
0	**pa** T T		**pb** A A		**pc** A A		**pd** A A		**pe** A A		**pf** A A	
5	**pg** A A		**ph** A A		**pi** A A		**pj** A A		**pk** A A		**pl** A A	
10	**pm** A A		**pn** A A		**po** A A		**pp** A A		**pq** A A		**pr** A A	
15	**ps** A A		**pt** A A		**pu** A A		**pv** A A		**pw** A A		**px** A A	
20	**py** A A		**pz** A A		**paa** A A		**pab** A A		**pac** A A		**pad** A A	

A: APART is the faster configuration
T: TOGETHER is the faster configuration

Figure 6-3:
Winning Configurations in Set P

Evidently the threshold between APART and TOGETHER is very far into the communication-intensive upper left-hand corner of the grid, within cell **pa**.

6-2.3. Experiment Set R

The intermediate grid used in experiment set P did not zoom in far enough to expose the transition from APART winning to TOGETHER winning. We proceed by creating a third grid which ‘‘zooms in’’ still closer on the high-communication, low-CPU, and low-I/O part of the process-characteristic space, the region defined by cell **pa**. We call this the *zoomed-in grid.* The levels of the two factors are:

CPU levels: 0%, 1%, 2%, 3%, 4%, and 5%
IO-COMM levels: 0/1, 1/99, 2/98, 3/97, 4/96, and 5/95

There are six levels of each factor, making 36 cases in the zoomed-in grid.

The two-process experiment set based on the zoomed-in grid is called experiment set R; see Figure 6-4. The four cells in the upper left corner of set P are duplicated in the four outside corner cells of set R. We ignore the duplication; new traces are generated and simulated for the corner cells of set R.

We have generated four traces for each cell in set R. The GENACTIONS program (see Section 5-3) uses random numbers in generating traces, so the actual trace characteristics may differ from the specified characteristics (the grid position) by 2-3%. The differences between specified characteristics and actual characteristics should be in random directions and of random magnitudes. If we generate enough traces for the same cell, the differences should cancel out, so that the collection of traces as a whole is a good representation of the grid position. We chose four repetitions as a reasonable trade-off between increasing the number of traces and keeping the experimental overhead low. With 36 cells in the grid and four traces per cell, there are 144 runs.

The winning, or faster, configuration for each run in set R is shown in Figure 6-4.

	IO-COMM					
CPU	IO 0 COMM 100	IO 1 COMM 99	IO 2 COMM 98	IO 3 COMM 97	IO 4 COMM 96	IO 5 COMM 95
0	**raa** T T T T	**rab** A T T T	**rac** T T T T	**rad** T A A T	**rae** A T T A	**raf** T T A A
1	**rag** A T T A	**rah** A A T T	**rai** A T A A	**raj** A T T A	**rak** A T A A	**ral** A A A T
2	**ram** A A A T	**ran** T A A A	**rao** A A A T	**rap** A A A A	**raq** A A A A	**rar** A A A A
3	**ras** A A A T	**rat** T A A A	**rau** A A A T	**rav** A A A A	**raw** A A T A	**rax** A A A A
4	**ray** A A A A	**raz** A A A A	**rba** A A A A	**rbb** A A A A	**rbc** A A A A	**rbd** A A A A
5	**rbe** A A A A	**rbf** A A A A	**rbg** A A A A	**rbh** A A A A	**rbi** A A A A	**rbj** A A A A

A: APART is the faster configuration
T: TOGETHER is the faster configuration

Figure 6-4:
Winning Configurations in Set R

There are regions where APART is always faster and regions where APART is not always faster, so set R will allow us to investigate the transition between these two areas.

6-2.3.1. The Transition Zone in Set R

In sets N and P, APART is the faster configuration in every case except the upper left corner. Even in set R, APART is faster in most of the runs. Clearly, in most of the space of program characteristics, the norm is that APART is the faster configuration.

The zoomed-in grid can be divided into two regions: the normal, all-APART region, in which APART is faster in every repetition of each case, and the not-all-APART region, in which APART is not the faster configuration for every run. We call this not-all-APART region the *transition zone.*† Figure 6-5 shows the grid cells which make up the transition zone in set R.

† We had originally planned to use the term transition zone to refer to the area of the zoomed-in grid between the all-TOGETHER region and the all-APART region. However, because the important division seems to be between all-APART and not-all-APART, and because the all-TOGETHER region is so small, only cell **raa**, we decided to focus on two regions rather than three, and to use the term transition zone anyway.

	IO-COMM											
CPU	IO 0	COMM 100	IO 1	COMM 99	IO 2	COMM 98	IO 3	COMM 97	IO 4	COMM 96	IO 5	COMM 95
0	**raa**		**rab**		**rac**		**rad**		**rae**		**raf**	
1	**rag**		**rah**		**rai**		**raj**		**rak**		**ral**	
2	**ram**		**ran**		**rao**				**raq**			
3	**ras**		**rat**		**rau**				**raw**			
4												
5												

Figure 6-5:
Cells Making Up the Transition Zone in Set R

Due to the random variation in actual process characteristics, as opposed to grid coordinates, the boundary of the transition zone is somewhat fuzzy. It may be that the two cells below **raj** (**rap** and **rav**) ''should be'' in the transition zone, that is, APART is not necessarily always faster in those cells, but in the traces used it does happen to be faster. It may be that **raw** should not be in the transition zone, that is, APART is always faster in traces with the characteristics of cell **raw**, but one of the **raw** traces happened to have the characteristics of a cell higher up in the grid. In fact, since APART is always best in **raq**, this is quite likely (see Figure 6-4).

6-2.4. Where Real Distributed Programs Fall in the Grid

We do not know what the characteristics of real distributed programs are, that is, we do not know where in the grids real programs fall. We believe that only programs which do substantial amounts of computational work will be turned into distributed programs. Whether done by hand or automatically, there is a certain overhead in creating a multi-process, distributable program, and the overhead will probably only be incurred for large programs. We also think that real programs will do little I/O. Most likely some parent process or master process will read in the data, then apportion it to parallel child or slave processes. The children will report their partial results back to the parent, which will write out the final results. Thus, we believe that real distributed programs are high-CPU and low-I/O, and fall into the lower left region of the full grid, the region occupied by cells **np**, **nq**, **nu**, **nv**, and **nz**. If our assumptions are highly accurate, perhaps almost all real distributed programs will fall into the region occupied by **nu** and **nz**.

We believe it is highly unlikely that any real distributed programs will fall into the upper left-hand part of the zoomed-in grid. That region represents extremely high communication, with very little computation or I/O. A group of distributed processes whose activities consist almost entirely of communicating with each other is not very useful. In fact it is probably a mistake and is not behaving as desired. Even if this behaviour is accomplishing something useful, it should probably be re-coded as a single process to reduce the communication overhead. Thus, for practical purposes, we do not believe that any useful real distributed programs will fall into the upper part of the zoomed-in grid.

We continue to study both the full grid and the zoomed-in grid, including the transition zone, because we wish to understand the factors which affect the response times of the configurations, and it is only in the transition zone that APART is not best.

6-3. Response Times

We now analyze the outcomes of experiment sets N and R in more detail. Set N, using the full grid, covers the space of process characteristics; set R, using the zoomed-in grid, focuses on the transition from TOGETHER winning to APART winning; there is no need to further examine set P. In the following sections, we examine the response times of the programs when run in each configuration, and the differences in the response times of the different configurations.

Let us use the term t_{APART} for the response time of the distributed program when executed in the APART configuration, from the moment the processes begin executing until the moment the last process finishes. Similarly, t_{TOG} is the response time of the TOGETHER configuration.

6-3.1. Response Times in Set N

Summary statistics on the response times of the 52 runs in experiment set N are shown in Figure 6-6.

Response Time Statistics (ms)							
Configuration	# Runs	Mean	Std. Dev.	Minimum		Maximum	
				Time	Case	Time	Case
APART	52	24,758	12,428	1,058	**na-1**	50,479	**nx-2**
TOGETHER	52	37,644	21,559	1,000	**na-1**	83,291	**nz-1**

Figure 6-6:
Summary Statistics on
Response Times in Set N

It is clear that, on average, APART is the better choice.

Figure 6-7 shows the mean response times for the runs in each row, for each configuration.

Mean Response Times by Row (ms)			
	mean t_{APART}	mean t_{TOG}	mean t_{TOG} − mean t_{APART}
Row 1	7,061	10,659	3,598
Row 2	15,987	20,302	4,315
Row 3	24,410	34,153	9,743
Row 4	32,373	48,363	15,990
Row 5	40,564	65,830	25,266
Row 6	41,748	82,208	40,460

Figure 6-7:
Mean Response Times by Row in Set N

There is a substantial increase in the response times from row to row. In set N, each row has 20% more execute actions than the preceding row. Each execute action requests an amount of CPU service drawn from a uniform random distribution between 2 ms and 200 ms, whereas I/O and communication actions usually take less time to carry out (see Section 5-3.7), so the larger the proportion of execute actions, the longer the response time.

The difference between the mean APART time and the mean TOGETHER time grows considerably larger with every row. The importance of choosing the faster configuration becomes more important the more CPU-intensive the program is.

There is some variation in the response times across the columns of the full grid, but the differences are much smaller than the differences between the rows, and there is no clear pattern. Recall, however, that the IO-COMM factor divides the non-execute actions. The rows closer to the top of the grid have

more non-execute actions, so we would expect the differences due to the IO-COMM balance to be most pronounced there. Figure 6-8 shows the response times of the runs in the top row.

Response Times of Individual Runs (ms)					
Config.	**na**	**nb**	**nc**	**nd**	**ne**
APART	1,058	5,130	7,006	11,176	10,380
	1,080	4,676	8,480	10,723	10,888
TOGETHER	1,000	6,173	9,693	15,767	18,979
	1,001	5,849	11,510	16,607	20,007

Figure 6-8:
Response Times of Runs in Row 1 of Set N

In row 1, the response times grow larger as one moves toward the right, toward more I/O actions and fewer communication actions. Evidently, I/O influences response time more than communication does.‡ In row 2, and even more so in lower rows, there is little pattern in the response times along the columns.

6-3.2. Response Times in Set R

Summary statistics on the response times of the 144 runs in experiment set R are shown in Figure 6-9.

Response Time Statistics (ms)							
Configuration	# Runs	Mean	Std. Dev.	Minimum		Maximum	
				Time	Case	Time	Case
APART	144	3,345	1,375	1,031	**raa-1**	6,838	**rbj-4**
TOGETHER	144	3,492	1,534	1,000	**raa-2**	7,572	**rbg-1**

Figure 6-9:
Summary Statistics on
Response Times in Set R

As opposed to set N, in which APART was clearly the better choice on average, it is not clear that APART is the better choice on average in set R. The mean response times of the two configurations are nearly the same, and the fastest and slowest times are also quite close. This is a reflection of the fact that in set R sometimes APART is faster and sometimes TOGETHER is faster. Indeed, we zoomed in on set R in order to achieve this.

In set R, as in set N, the response times of both configurations increase as one moves down the rows of the grid, as shown in Figure 6-10.

‡ Of course, the simulation parameters specify the I/O and communication service times, but only through simulation can we tell the elapsed times and the effect of the elapsed times on the relative speeds of the configurations.

Mean Response Times by Row (ms)			
	mean t_{APART}	mean t_{TOG}	mean t_{TOG} − mean t_{APART}
Row 1	1,504	1,471	-33
Row 2	2,290	2,304	14
Row 3	2,986	3,073	87
Row 4	3,659	3,801	142
Row 5	4,372	4,616	244
Row 6	5,257	5,688	431

Figure 6-10:
Mean Response Times by Row in Set R

The increases are smaller, even percentage-wise, in set R than in set N, as are the differences between the mean response times of the configurations. In set N each row has 20% more execute actions, while in set R each row has only 1% more execute actions.

6-3.3. Summary

We have seen that the response times of both configurations rise as the proportion of execute actions grows. In both experiment sets, the sizes of the differences in the row means are larger in lower rows of the grids, meaning that as CPU use increases, TOGETHER becomes slower much more quickly than APART does. We have also seen that communication actions are faster than I/O actions. The APART configuration is faster on average in set N, but in set R the two configurations have similar mean response times.

6-4. Differences between TOGETHER and APART Response Times

Our goal is to select the faster configuration, but if the configurations have nearly the same response times it may not be worth the time it takes to identify the slightly faster configuration. More generally, we would like to know how much faster the response time of the faster configuration is compared to the slower, that is, the difference between the response times of the two configurations. We would like to find the mean response time difference over the two experiment sets, as well as the maximum difference.

Let us define t_{diff} to be $t_{TOG} - t_{APART}$. Since APART is usually the winner, t_{APART} is usually smaller than t_{TOG}, so t_{diff} is usually a positive number. We have computed t_{diff} for each run in sets N and R. Summary statistics on t_{diff} in sets N and R are shown in Figure 6-11.

Response Time Differences t_{diff} (ms)							
Set	# Runs	Mean	Std. Dev.	Minimum		Maximum	
				t_{diff}	Case	t_{diff}	Case
Set N	52	12,885	10,003	-80	**na-2**	41,316	**nz-1**
Set R	144	147	205	-154	**raa-3**	1,003	**rbg-1**

Figure 6-11:
Summary Statistics on
Response Time Differences t_{diff}
in Sets N and R

As expected, the differences are much larger in set N than in set R. The standard deviations are large relative to the means, indicating that there are some large differences, and indeed, the maxima are very large.

Differences may be positive or negative. When computing the mean, the positive and negative values cancel each other out to some extent, so that the mean does not give us the typical size of individual values. The magnitude of the response time difference $|t_{diff}|$, indicates how important it is to find

the best configuration. Computing the means of the magnitude of the differences gives us the average magnitudes of the response time differences; these means and other summary statistics for sets N and R are shown in Figure 6-12.†

Magnitudes of Response Time Differences $\lvert t_{diff} \rvert$ (ms)							
Set	# Runs	Mean	Std. Dev.	Minimum		Maximum	
				$\lvert t_{diff} \rvert$	Case	$\lvert t_{diff} \rvert$	Case
Set N	52	12,891	9,996	58.32	**na-1**	41,316	**nz-1**
Set R	144	173	184	0.72	**rag-1**	1,003	**rbg-1**

Figure 6-12:
Summary Statistics on
Magnitude of Response Time Differences $\lvert t_{diff} \rvert$
in Sets N and R

In set N, the mean difference (mean t_{diff}) is very close to the average magnitude of the difference (mean $\lvert t_{diff} \rvert$) because there are only two negative differences in set N (the **na** runs). In set R, the average magnitude of t_{diff} is slightly larger than the mean t_{diff}, but still does not indicate that it makes more than a small difference which configuration is used. The difference between the mean value and the average magnitude is larger in set R because there are more negative differences in set R. As before, the standard deviations are sizable relative to the means, indicating that there are some large values.

6-5. Relative Differences in Response Times

The differences in response times discussed above are specific to our traces. To generalize the results, we consider the size of t_{diff} relative to the response times. We compute the *relative percentage difference* using the formula

$$[(t_{TOG} - t_{APART})/t_{TOG}] \cdot 100$$

We use the TOGETHER response time in the denominator, so the relative percentage difference is also the percentage improvement in response time obtained by using APART instead of TOGETHER. Since APART is usually the faster configuration, the relative percentage difference is usually positive.

Based on the pattern of winning configurations we have seen, we expect to find the largest differences in set N and the smallest differences in set R. We also expect to find that within each grid, the differences are largest in the lower right and get smaller as one moves toward the upper left.

Summary statistics on the relative percentage differences in set N and set R are shown in Figure 6-13. Also shown are summary statistics on the magnitudes of the relative percentage differences.

† The standard deviation, the computation of which involves squaring then taking the square root, provides somewhat similar information, but we wanted to examine the actual average magnitude of the differences.

Relative Percentage Differences						
Set	# Runs	Relative Percentage Differences				
			Mean	Std. Dev.	Minimum	Maximum
Set N	52	Actual Value	29.63	11.40	-7.96	49.60
		Magnitude	30.16	9.92	5.83	49.60
Set R	144	Actual Value	2.73	4.88	-15.33	14.55
		Magnitude	4.47	3.35	0.02	15.33

Figure 6-13:
Summary Statistics on
Relative Percentage Differences
in Set N and Set R

Figures 5-14 and 5-15 show the relative percentage differences between the response times for APART and TOGETHER for each run in sets N and R.

	IO-COMM									
CPU	IO 0	COMM 100	IO 25	COMM 75	IO 50	COMM 50	IO 75	COMM 25	IO 100	COMM 0
0	**na**		**nb**		**nc**		**nd**		**ne**	
		-5.83		16.88		27.72		29.12		45.31
		-7.96		20.05		26.24		35.43		45.58
20	**nf**		**ng**		**nh**		**ni**		**nj**	
		17.84		17.79		18.83		18.66		29.55
		23.94		15.17		22.74		19.86		26.15
40	**nk**		**nl**		**nm**		**nn**		**no**	
		24.06		24.70		27.34		25.55		30.75
		31.57		30.81		26.60		28.03		35.14
60	**np**		**nq**		**nr**		**ns**		**nt**	
		36.36		33.10		31.31		24.69		39.21
		32.84		35.20		34.50		22.74		40.50
80	**nu**		**nv**		**nw**		**nx**		**ny**	
		36.57		38.01		37.87		36.34		45.80
		35.92		43.37		38.49		25.88		45.82
100	**nz**									
		49.60								
		48.82								

Figure 6-14:
Relative Percentage Differences in Set N

	IO-COMM											
	IO	COMM	IO	COMM	IO	COMM	IO	COMM	IO	COMM	IO	COMM
CPU	0	100	1	99	2	98	3	97	4	96	5	95
0	**raa**		**rab**		**rac**		**rad**		**rae**		**raf**	
		-3.09		1.34		-8.50		-0.81		0.36		-2.61
		-7.20		-2.09		-2.21		1.59		-7.37		-3.87
		-15.33		-12.63		-7.66		7.06		-0.58		0.30
		-8.67		-9.51		-4.64		-3.81		7.61		5.85
1	**rag**		**rah**		**rai**		**raj**		**rak**		**ral**	
		0.04		0.19		1.09		3.14		2.84		4.54
		-3.72		2.14		-1.81		-0.04		-3.81		2.89
		-2.59		-1.90		2.58		-2.14		2.79		1.17
		0.80		-3.10		4.98		2.63		0.76		-1.00
2	**ram**		**ran**		**rao**		**rap**		**raq**		**rar**	
		5.68		-0.07		0.86		0.77		2.87		2.82
		7.99		4.86		5.88		4.97		2.79		0.88
		0.39		0.02		2.51		0.71		4.62		2.75
		-0.12		4.41		-0.57		4.62		7.34		1.64
3	**ras**		**rat**		**rau**		**rav**		**raw**		**rax**	
		4.81		-0.12		4.22		3.37		3.72		7.13
		2.10		4.09		1.96		5.02		5.61		7.83
		7.17		11.08		4.13		1.41		-0.62		2.49
		-0.34		1.39		-2.81		7.66		0.70		3.33
4	**ray**		**raz**		**rba**		**rbb**		**rbc**		**rbd**	
		7.98		6.35		7.03		4.88		1.92		7.89
		6.86		4.94		6.41		10.51		8.37		3.89
		2.01		2.89		1.13		0.98		5.50		5.39
		10.22		2.49		5.64		8.76		1.80		3.78
5	**rbe**		**rbf**		**rbg**		**rbh**		**rbi**		**rbj**	
		14.55		12.04		13.24		4.80		4.13		6.68
		6.58		6.93		6.34		3.37		1.95		3.69
		3.92		5.42		5.99		8.36		4.91		10.54
		13.48		10.42		4.55		5.31		9.85		9.08

Figure 6-15:
Relative Percentage Differences in Set R

As expected, the differences are much larger in set N than in set R. In set R, most of the differences are just a few percentage points; only a few are greater than 10%. Also as expected, within each grid the differences were larger in the lower right and smaller toward the upper left. This confirms our hypothesis that the relative differences get smaller as one moves further into the upper left.

The sizes of the relative differences range from close to zero in set R to almost 50% in set N. The largest is 49.60% in case **nz-1**. In **nz-1**, the actual response times are 41,975 ms in the APART configuration and 83,291 ms in TOGETHER, so the 50% difference is on the order of 40 seconds. In a large job the amount of time saved could be substantial; for example, a job running for eight hours could be cut down to four hours by choosing the right configuration. This is clearly a substantial difference, and it indicates that the potential benefit of finding the best configuration is quite high in at least some distributed programs.

Examining Figures 5-14 and 5-15, we see that there is considerable variation in the relative percentage differences. It appears that differences in lower rows are generally larger than those in higher rows. The mean relative percentage differences for each row in sets N and R are shown in Figure 6-16.

Set N		
Row	# Runs	Row Mean
Row 1	10	23.25%
Row 2	10	21.05%
Row 3	10	28.45%
Row 4	10	33.05%
Row 5	10	38.41%
Row 6	2	49.21%

Set R		
Row	# Runs	Row Mean
Row 1	24	-3.19%
Row 2	24	0.52%
Row 3	24	2.86%
Row 4	24	3.56%
Row 5	24	5.32%
Row 6	24	7.34%

Figure 6-16:
Mean Relative Percentage Differences by Row

The row means are indeed considerably larger toward the bottoms of the grids in both experiment sets. Again we see that the TOGETHER response time grows more quickly than the APART response time as one moves down the rows to increasingly CPU-intensive programs. We saw this in Section 6-3, above, when we saw the size of the differences between the APART and TOGETHER row mean response times grow larger in the lower rows.

It also appears that the differences in the right-most column of each row are larger than the other differences in the row. Evidently no communication somehow leads to large differences.

We do not know where in the grid real distributed programs fall. When we calculate the mean difference over the grid we are giving each cell equal weight. If real distributed programs fall in all parts of the full grid uniformly, then the average difference is the 30% calculated. If real programs fall more frequently into some parts of the grid we should give those parts greater weight in calculating the mean. If real programs fall only into certain parts of the grid we should look only at those parts of the grid.

We believe that real distributed programs fall mostly into the lower left part of the full grid, into the region of high CPU and low I/O (see Section 6-2.4). The differences in that region, and therefore probably the differences to be found in real distributed programs, are in the range 35-50%; see Figure 6-14.

The 4.5% average magnitude of the differences in set R is not as significant as the 30% average magnitude in set N, but is still substantial, so that even in the zoomed-in grid where it makes least difference which configuration is used, it can still be worth the cost of identifying the fastest configuration. In a job which runs for 10 hours in the worst configuration, a 4.5% improvement is 27 minutes. The time cost of selecting the faster configuration should be only a tiny fraction of that.

In set R, as in set N, the relative percentage differences grow larger row by row as one moves down the rows of the zoomed-in grid, as shown in Figure 6-16.

In summary, the more compute-intensive a program is, the larger the difference in response times between the two configurations. The mean relative percentage difference over the set N full grid is about 30%. In the lower left region, into which we expect most real distributed programs to fall, the differences are in the range 35%-50%.

6-6. The Always-APART Policy

6-6.1. Introduction

In nearly all of the full grid, the APART configuration is faster than the TOGETHER configuration. Even in the zoomed-in grid, APART is more often the faster configuration. This presents the possibility of using a very simple placement policy: always use the APART configuration. In the following sections we examine this always-APART policy. In Section 6-6.2 we discuss the pros and cons of the always-APART policy, in Section 6-6.3 we analyze the success of the policy and the penalty when it fails, and in Section 6-6.4 we present our conclusions concerning always-APART.

6-6.2. Pros and Cons

The most important advantage of the always-APART policy is that it requires no time to make a placement decision at the start of execution. Thus always-APART is a zero-cost policy in terms of our most important measure, response time. Even if other, more complex placement policies result in better placement decisions and faster response times, these advantages will be at least partially offset by the longer times required to make the placement decision before execution of the processes can begin. This decision-making time would partially offset the faster response time, making the net speedup smaller than the apparent speedup. The decision-making times might actually be larger than the differences in response times, so that the net results are *slower* response times. (If placement decisions can be pre-computed this argument does not apply.)

Another advantage of the always-APART policy is that no information on which to base the placement decision would be needed. This would save the costs of collecting data about the behaviour of the processes, either manually or automatically. It would save the costs of storing the data. It would avoid the problem of determining when to update it, and the effects of using inaccurate or out-of-date data. It would save the time-cost of accessing the information at placement time.

Yet another point in favor of the always-APART policy is that it would be extremely easy to implement, thereby reducing programming overhead, and reducing the possibility of errors in the code. It is difficult to imagine that any other policy could cost so little in any of these respects, much less in all respects.

The most important disadvantage of the always-APART policy is that it does not always select the fastest configuration. If the APART configuration is used when another configuration would be faster, the program will not execute as quickly as possible. This slow-down is at least partially offset by the fast decision time of the always-APART policy.

Another possible disadvantage of always-APART is that it always places processes on both processors. If it is important to use the minimum number of processors, then always-APART is at a disadvantage. This is more likely to be important when more than two processes are involved. It is also possible that there might be a greater overhead in dispatching a process to a remote processor, which we would incur needlessly if we used the APART configuration when another configuration would be faster.

To fully evaluate the always-APART policy we must determine how often always-APART would choose the wrong configuration, and how much the response time would suffer when the wrong choice is made. We present that analysis in the following section.

6-6.3. Analysis

In most runs in both experiment sets, APART is the winner, that is, the response time of the distributed program when executed in the APART configuration is faster than the response time when executed in the TOGETHER configuration. In set N, APART is faster in all runs except the two in the upper left-hand

corner cell; APART wins in 50 of the 52 runs, or 96.15% of the runs. In set R, APART wins in 111 out of 144 runs, or 77.08% of the runs; TOGETHER wins 33 times, or 22.92%.

If we use the APART configuration when TOGETHER is the faster configuration, we pay a penalty in the form of a slower-than-necessary response time. The penalty is the difference between the slower response time of APART and the faster response time of TOGETHER. We compute this *APART penalty* for each run in set R as follows. When APART is the winner the penalty is zero; when TOGETHER is the winner, the penalty is $t_{APART} - t_{TOG} = -1 \cdot t_{diff}$ but since we only take this difference when t_{APART} is larger, $-1 \cdot t_{diff}$ is always positive and $-1 \cdot t_{diff} = |t_{diff}|$.

In set R, the mean APART penalty is 12.66 ms. If we followed the always-APART policy and always used the APART configuration, and if programs fall uniformly across the zoomed-in grid, on average our programs would run 12.66 ms slower than they would if we always used the faster configuration. The mean of all APART and TOGETHER response times in set R is 3,418 ms, so the mean APART penalty of 12.66 ms is only 0.37% of this mean response time. The largest penalty is 153.99 ms, which is 4.50% of the mean response time. The mean APART penalty in set R is a small amount of time, and a very small percentage of the mean program response time. Even the maximum penalty is fairly small.

These computations implicitly assume that equal weight should be given to each cell in the zoomed-in grid used in set R, so that misses and penalties are distributed evenly over the grid. In fact, the cells in which always-APART misses are concentrated in the upper part of the grid, and to some extent on the left-hand side of the grid. Furthermore, the largest penalties occur in the uppermost and leftmost parts of the grid. If many real programs fall into the upper left region of the grid, more weight should be given to that region and the penalties will be higher. If more real programs fall into lower parts of the zoomed-in grid, or into the full grid, where always-APART is more accurate, the miss rate and the mean penalty will be smaller.

The upper part of the zoomed-in grid is where all the TOGETHER wins occur. For practical purposes, then, the hit rate of the always-APART policy approaches 100%, and the mean penalty approaches zero. Only in programs so unusual as to be pathological will there be any penalty at all.

6-6.4. Conclusions

We said above that the most important disadvantage of the always-APART policy is that APART is slower than TOGETHER in some parts of the grid. We have seen that real programs are unlikely to fall into those parts of the grid, so there are almost certain to be no misses with always-APART. The only remaining objections to always-APART are that it may be advantageous to use the smallest number of processors, or to reduce remote process creation. We believe that these objections are minor, and that the always-APART policy is clearly advantageous.

Having found a good placement policy, we could conclude our research into the 2x2 situation at this point. There are, however, a number of reasons it is worth continuing our explorations. First of all, we wish to understand the factors which influence the response times of distributed programs. We believe that we can gain this insight by further analysis of the simulations. Second, although always-APART is a good policy for 2x2, it may not be good for 3x3 or larger situations. Since it is probably easier to understand the workings of the 2x2 situation than the 3x3, we chose to analyze 2x2 before working on the 3x3 situation. Finally, there may not always be enough processors available to use the APART configuration, especially in larger situations such as 8x8. In such a case, some processes will have to be assigned to the same processor. In order to decide which processes to collocate, we must have an understanding of the factors that influence response times. Thus, in the following sections, and indeed in the rest of the thesis, we continue to analyze the behaviour of distributed programs.

6-7. Statistical Analyses

6-7.1. Introduction

Our purpose is to identify the factors that have strong influences on which of the configurations is fastest. More precisely, we are interested in the characteristics of the program, the data, and the system that influence the size and sign of the relative difference between the response time of the APART configuration and the response time of the TOGETHER configuration. We want to learn how to use these characteristics to predict which configuration will be fastest, and how large the difference between the response times of the configurations will be.

In Section 6-7.2, we describe the variables drawn from the data and used in the analyses. In Section 6-7.3, we describe the analyses concerning selecting the faster configuration, and their results. Section 6-7.4 is a slight digression; we discuss how much of processes' communication waiting time is due to communication delay and how much to synchronization.

6-7.2. The Variables

In our experiments, artificially generated traces represent the program and typical input data; a summary of each trace is produced by the SUM program. The system characteristics are described in the Carnival simulator's parameter file. Carnival simulates the execution of the program on the input using the described system characteristics, and produces output which describes the execution behaviour. (See Chapter 5 for more information about Carnival and SUM.)

These trace and system characteristics are represented by *variables*. Each variable takes on a particular value for each run, and the values for all the runs in an experiment set are a *data set.* Data set N corresponds to experiment set N, and data set R corresponds to experiment set R.

The first group of variables, described in Section 6-7.2.1, below, are those taken from the trace summaries. If it were possible to make good placement decisions based only on information from the traces, those placement decisions would be correct for any system characteristics and any load environment, but it is probably not possible.

Since the system characteristics are identical for all simulation runs, they will not explain variability in the configuration response times. The system characteristics act indirectly, through their effect on the execution behaviour.

The second group of variables, described in Section 6-7.2.2, below, are those taken from the execution behaviour, as reported in the simulation output.

The third group of variables, described in Section 6-7.3.3, below, are ratios of computation to communication.

6-7.2.1. Summary Variables

A summary of each trace is produced by the SUM program (described in Section 5-5), and the data from the summaries for each run are used as variables. The names of these variables start with ''*s,*'' for ''summary variable.'' There are two processes, numbered 0 and 1. For process 0, the variable names begin with ''*s0.*'' Figure 6-17 lists the *s0* summary variables.

Variable Name	Description of Variable
s0actcnt	count of all actions
s0excnt	count of execute actions
s0extime	sum of execute times (ms)
s0iocnt	count of I/O actions, including both reads and writes
s0iosize	sum of I/O data sizes (bytes)
s0rdcnt	count of read actions
s0rdsize	sum of read data sizes (bytes)
s0wrcnt	count of write actions
s0wrsize	sum of write data sizes (bytes)
s0sndcnt	count of send actions
s0sndsiz	sum of send data sizes (bytes)
s0rcvcnt	count of receive actions
s0tercnt	count of terminate actions (should be 1)
s0expct	percent of actions that are executes
s0iopct	percent of actions that are I/Os
s0srpct	percent of actions that are sends or receives
s0terpct	percent of actions that are terminates
s0iocio	in the IO/COMM division, the proportion that is I/O
s0ioccom	in the IO/COMM division, the proportion that is COMM

Figure 6-17:
Summary Variables for Process 0

The summary variables for process 1 have identical names, except that they begin with ‘‘*s1,*’’ for example, *s1actcnt*, *s1excnt*, *s1extime*, and so on. Their meanings are identical to those of the corresponding process 0 variables shown in Figure 6-17.

To characterize each program as a whole, we computed the mean of each summary variable over both processes. For example, the mean action count for the trace is the mean of *s0actcnt* and *s1actcnt*. These summary mean variables begin with ‘‘*sm,*’’ for example, *smactcnt*, *smexcnt*, *smextime*, and so on.

6-7.2.2. Execution Behaviour Variables

Execution behaviour variables are variables whose values are measured during simulation. The most important of these are the dependent variables t_{APART}, t_{TOG}, and t_{diff}.

Independent execution behaviour variables measured during simulation include the CPU service time, the I/O service time, and the communication waiting time of each process in each configuration. From these per-process values we compute the mean values over the two processes, for each configuration.

These independent execution behaviour variables have names beginning with ‘‘*af*’’, for ‘‘actual values corresponding to formula (computed) values.’’ Following the ‘‘*af*’’ is a ‘‘*0,*’’ a ‘‘*1,*’’ or an ‘‘*m,*’’ depending on whether the variable represents a value from process 0, process 1, or the mean of the two processes, respectively. The CPU service requirement variables have names containing ‘‘*cr,*’’ the I/O service time variables have names containing ‘‘*i,*’’ and the waiting time variables have names containing ‘‘*w.*’’ Finally, ‘‘*ap*’’ indicates values taken from APART simulations and ‘‘*tog*’’ indicates values taken from TOGETHER simulations. For example, *af0crap* is the CPU requirement (CPU service time) of process 0 in the APART configuration, *afmitog* is the mean I/O service time of the two processes in the TOGETHER configuration (computed as (*af0itog*+*af1itog*)/2). The complete list of *af* variables is shown in Figure 6-18.

Variable Name for Process 0	Variable Name for Process 1	Variable Name for Mean of Procs.	Description
Values taken from APART simulation outputs:			
af0crap	*af1crap*	*afmcrap*	actual CPU service requirement
af0iap	*af1iap*	*afmiap*	actual I/O service requirement
af0wap	*af1wap*	*afmwap*	actual communication waiting time
Values taken from TOGETHER simulation outputs:			
af0crtog	*af1crtog*	*afmcrtog*	actual CPU service requirement
af0itog	*af1itog*	*afmitog*	actual I/O service requirement
af0wtog	*af1wtog*	*afmwtog*	actual communication waiting time

Figure 6-18:
The *af* Execution Behaviour Variables
from the Simulation Outputs

Another execution behaviour variable placed in the data sets is *aspf*, the actual value of SPF. The SPF is discussed in Section 6-7.4, below.

6-7.2.3. Computation-to-Communication Ratios (CCRs)

Intuitively, one would think that a ratio of computation to communication would be important. The more computation, the greater the advantage of using more processors to do the computation. The more communication, the greater the advantage of placing multiple processes on the same processor to avoid long inter-processor communication delays. The more computation relative to communication, the less processors will be slowed down by having to wait for messages to arrive. We defined a number of computation-to-communication ratios, or *CCRs*, which became variables in the data sets. We expected that the statistical analyses would tell us that at least some of them were useful.

In computing most of the CCRs, the amount of computation and the amount of communication are measured by the amounts of time spent on those activities. The amounts of time actually spent are part of the execution behaviour and are available in the simulation outputs. One of our objectives, however, is to see how successful we can be at identifying the best configuration using the least information, without simulating or running the program. Thus, the CCRs are computed using estimates of the computation and communication times, estimates based on the trace summaries, some system parameters, and certain simplifying assumptions. If the CCRs computed using estimates are useful in identifying the fastest configuration, only a small and well-defined amount of system information will be needed in making placements. This could make placement decisions more general and more stable than if more system information were needed, or if it were not clear which system characteristics were significant.

The CCRs and the formulas used to compute them are shown in Figure 6-19. In addition to computing *CCR1* and *CCR2*, we also computed their inverses, *CCR1inv* and *CCR2inv*.

Original	Inverse
$CCR1 = \frac{CR + I}{rcvCount}$	$CCR1inv = \frac{rcvCount}{CR + I}$
$CCR2 = \frac{CR + I}{C}$	$CCR2inv = \frac{C}{CR + I}$
$CCR3 = \frac{smextime}{smrcvcnt}$	

Figure 6-19:
CCRs: Computation/Communication Ratios

In the following sections we introduce the terms used in computing CCR1 and CCR2. The factors of CCR3, *smextime* and *smrcvcnt*, are trace summary variables described in Section 6-7.2.1, above.

6-7.2.3.1. Estimates of Computation used in the CCRs

One estimate of computation is the amount of time spent doing all non-communication activities, that is, CPU computation and I/O. CPU time is captured in the quantity *CR*, the CPU service time requirement; I/O service time is captured in the quantity *I*. In this section we first describe the computation of *CR*, then the computation of *I*.

The estimated total CPU service time requirement of process *i* is CR_i. This includes the CPU time explicitly requested in execute actions; CPU time is also used whenever I/O, send, or receive actions are performed. The CPU overheads incurred by each type of action are system parameters. The names of the parameters and the parameter values used in our experiments are listed in Figure 6-20.

Parameter	Description	Value
	CPU time required to:	
io_cpu_time	perform an I/O operation	2.0 ms
send_rmt_cpu_time	send a message to a process on a remote processor	2.0 ms
send_lcl_cpu_time	send a message to a process on the local processor	1.5 ms
rcv_cpu_time	receive a message	1.0 ms

Figure 6-20:
CPU Service Time
Required for Each Action Type

The configuration determines which messages are local and which are remote, while the CCRs are estimates independent of configuration, so we do not know which messages are local and which remote. To compute *CR*, we assume that all messages are remote; this is equivalent to the APART configuration. Since *send_rmt_cpu_time* is larger than *send_lcl_cpu_time*, this gives us the maximum *CR*.

The CPU requirement of process *i*, CR_i, is

$$CR_i = extime_i + (sendCount_i \cdot send_rmt_cpu_time) + (receiveCount_i \cdot rcv_cpu_time) + (ioCount_i \cdot io_cpu_time)$$

The total *CR* for the program is the sum of the CR_i for all processes, $CR = \sum_i CR_i$.

The CPU requirements of the individual processes, CR_0 and CR_1, are part of the data sets under the variable names *f0cr* and *f1cr*; the *f* stands for ''computed formula.'' The mean per-process CPU requirement $fmcr = (f0cr + f1cr)/2$ is also part of the data sets.

The time required to perform all the I/O operations for process *i* is

$$I_i = (readCount_i \cdot io_read_time) + (writeCount_i \cdot io_write_time)$$

where $readCount_i$ and $writeCount_i$ are the number of reads and writes performed by process *i*, respectively. To simplify the computation of *I* we pretend that every read operation reads one block of data. The *io_read_time* is then one latency time plus the time to transfer one block of data:

$$io_read_time = io_latency + io_transfer$$

Io_latency and *io_transfer*, as well as the size of an I/O block, *io_blocksize*, are system parameters. The parameters values we have used are listed in Figure 6-21.

Parameter	Description	Value
io_blocksize	number of bytes in an I/O block of data	8192 bytes
io_latency	time to start an I/O operation	30.0 ms
io_transfer	time to transfer one block of data to or from the I/O subsystem	8.0 ms

Figure 6-21:
I/O Parameters and Values

In Carnival, write operations do not require any I/O service; it is assumed that data is written to an in-memory disk cache, to be written to disk later, asynchronously. The time to copy data to the disk cache is assumed to be part of *io_cpu_time*, thus *io_write_time* is zero (see Section 5-2.4.3.2). We use these same assumptions in computing the I_i. Thus,

$$I_i = readCount_i \cdot (io_latency + io_transfer)$$

and the total I/O time for the program is the sum of the I_i for all processes, $I = \sum_i I_i$.

6-7.2.3.2. Estimates of Communication used in the CCRs

We now turn to the estimates of communication used to compute the computation-to-communication ratios.

The numbers of receive actions performed by processes 0 and 1 are *s0rcvcnt* and *s1rcvcnt*, respectively. Let us call the sum of these *rcvcnt*, and the mean *smrcvcnt*.

The estimated communication delay, *C*, is the time messages spend in transit from the sending process to the receiving process' incoming message queue. For local messages (sending and receiving processes on the same processor) the transit time is zero. For remote messages, the transit time depends on the length of the message. Because we want to make simple estimates, without knowing which configuration is to be used, we make two assumptions: (1) all messages are to remote destinations, and (2) all messages are one packet long. We compute the remote communication delay of one packet, *packet_delay*, as one latency time plus the transmission time of one packet:

$$packet_delay = comm_latency + (packet_size/cbw)$$

where *cbw* is the communication bandwidth in bytes/ms, which is derived from the parameter *comm_bandwidth* in megabits per second. The communication delay for process *i* is estimated as

$$C_i = receiveCount_i \cdot packet_delay$$

The total communication delay for the program is $C = \sum_i C_i$.

In computing the CCRs, if the denominator is zero the CCR is given the value zero.

Having described the variables used in the statistical analyses, we turn, in the following section, to describing the analyses and their results.

6-7.3. Predicting t_{diff}

6-7.3.1. Introduction

We used the correlation, regular regression, and stepwise regression procedures of the SAS statistics package [SAS85a, SAS85b] to do the statistical analyses. Data sets N and R were created, corresponding to experiment sets N and R, with one record for each run, containing the variables described above. All analyses were run separately on data set N and data set R.

In the regular regressions and the stepwise regressions we used three dependent variables: t_{APART}, t_{TOG}, and t_{diff}. Many variables made very good predictions of t_{APART} and t_{TOG}. It was much harder to predict t_{diff}, because t_{diff} is considerably smaller than the two response times, and also because it is very sensitive to inaccuracies in predicting either t_{APART} or t_{TOG}. Because t_{diff} is the key to choosing which configuration to use, we concentrate on it in the following. The independent variables used differ from one analysis to another, and are described for each analysis.

The most useful measure of success in the regressions is the coefficient of determination, R^2. The values R^2 takes on are $0 \le R^2 \le 1$; as Allen puts it, ''if R^2 is close to one, the regression model is a good predictor while if it is near zero the regression model is a poor predictor'' [All90, p. 594]. More specifically, ''an R^2 value of 0.667 is considered evidence that the values of y can be predicted ... using the regression .... For scientific or engineering problems, however, we like to see R^2 values of at least 0.8'' [All90, p. 571].

We also use the t ratios for the null hypotheses that each independent variable in a regression has no significant effect on the regression model. We follow the rule that $|t| > 2.0$ indicates that the variable is very likely to be significant. In figures showing the results of stepwise regressions, the F values shown for individual variables are the F values to enter the regression model.

6-7.3.2. The Reduced Set of Seventeen Variables

Many of the variables in the data sets are highly correlated. We used SAS to compute the correlations between the variables within each of the data sets N and R, and identified groups of variables which are highly correlated to each other. We picked one variable to represent each group using a somewhat subjective procedure. The resulting list of 17 variables is shown in Figure 6-22.

Variable Name	Description of Variable
s0sndcnt	count of send actions in process 0
s0sndsiz	sum of send data sizes (bytes) in process 0
s0rcvcnt	count of receive actions in process 0
s1sndcnt	count of send actions in process 1
s1sndsiz	sum of send data sizes (bytes) in process 1
s1rcvcnt	count of receive actions in process 1
smextime	mean sum of execute times (ms)
smrdcnt	mean count of read actions
smwrcnt	mean count of write actions
smsndcnt	mean count of send actions
smsndsiz	mean sum of send data sizes (bytes)
smactcnt	mean count of all actions
smterpct	mean percent of actions that are terminates
smiocio	in the IO/COMM division, the mean proportion that is I/O
smioccom	in the IO/COMM division, the mean proportion that is COMM
CCR1	$\frac{CR+I}{rcvCount}$
CCR1inv	$\frac{rcvCount}{(CR+I)}$

Figure 6-22:
Reduced Set of 17 Variables

For the most part, the correlations among the variables in this set are low, however, there are exceptions. Some of the variables have a low correlation in one data set but a high correlation in the other. Some variables were included despite their high correlations with other variables because we expected them to be important and wanted to see how they fared in the analysis.

6-7.3.3. Analyzing the Set of Seventeen Variables

We ran stepwise regressions and regular regressions to see which of the 17 independent variables were most significant.

The results of stepwise regression of t_{diff} using the reduced set of 17 variables are shown in Figure 6-23.

Set N			
total R^2=0.9486			
overall $F(7,44)$=115.9, (prob>F)=0.0001			
Variable	Partial R^2	F	prob>F
smextime	0.8615	310.9	0.0001
smioccom	0.0289	12.9	0.0007
smiocio	0.0274	16.0	0.0002
smwrcnt	0.0063	4.3	0.0426
smactcnt	0.0056	4.1	0.0492
smterpct	0.0067	5.4	0.0251
CCR1inv	0.0033	2.8	0.0994

Set R			
total R^2=0.6638			
overall $F(3,140)$=92.1, (prob>F)=0.0001			
Variable	Partial R^2	F	prob>F
smextime	0.6483	261.7	0.0001
CCR1	0.0057	2.3	0.1299
smrdcnt	0.0098	4.1	0.0458

Figure 6-23:
Stepwise Regression of t_{diff}
using 17 Independent Variables

The large differences in partial R^2 values are not as significant as they appear. The earlier-selected variables usually have larger R^2; if the order in which variables were selected were different, the partial R^2 values would be much different. Nevertheless, the order in which variables are selected and the partial R^2 values can give useful insights. For instance, we clearly see that *smextime* is important in both sets N and R.

The results of regular regression of t_{diff}, again using the reduced set of 17 independent variables, are shown in Figure 6-24. Only the variables with $|t|>2.0$ are shown.

Set N		
R^2=0.9590		
$F(15,36)$=56.2, (prob>F)=0.0001		
Variable	t	prob>|t|
smiocio	-3.7	0.0008
smactcnt	-3.6	0.0009
smextime	3.3	0.0020
smioccom	-2.6	0.0121
smterpct	-2.3	0.0265
smrdcnt	2.2	0.0383

Set R		
R^2=0.7415		
$F(14,129)$=26.4, (prob>F)=0.0001		
Variable	t	prob>|t|
smioccom	5.6	0.0001
smwrcnt	5.5	0.0001
smrdcnt	5.0	0.0001
CCR1	4.6	0.0001
smextime	-4.4	0.0001
smterpct	-2.4	0.0187
smactcnt	-2.3	0.0206
s1sndcnt	-2.1	0.0337

Figure 6-24:
Regular Regression on t_{diff}
using 17 Independent Variables

6-7.3.4. The Further Reduced Set of Four Variables

Based on the results of the regressions on 17 variables, we selected *smextime*, *smrdcnt*, *CCR1*, and *smioccom* as a further reduced set of variables. In set R, there appear to be five most significant variables, the four selected and *smwrcnt*. We felt that *smrdcnt* represented much the same information as *smwrcnt*. In set N, there appear to be six most significant variables, of which two pairs seem to represent similar information, leaving four groups: *smactcnt* and *smterpct*, *smiocio* and *smioccom*, *smextime*, and *smrdcnt*. The four variables selected include three of these six most significant variables for set N, and represent three of the four groups. The correlations among the four variables are shown in Figure 6-25.

Set N				
	smextime	*smrdcnt*	*CCR1*	*smioccom*
smextime	1.00			
smrdcnt	-0.56	1.00		
CCR1	0.49	-0.26	1.00	
smioccom	-0.12	-0.66	-0.03	1.00

Set R				
	smextime	*smrdcnt*	*CCR1*	*smioccom*
smextime	1.00			
smrdcnt	-0.02	1.00		
CCR1	0.97	0.23	1.00	
smioccom	0.003	-0.98	-0.24	1.00

Figure 6-25:
Correlation Coefficients
among the Four Variables

The results of stepwise regression and regular regression on t_{diff} using the four independent variables are shown in Figures 5-26 and 5-27.

Set N			
total R^2=0.9133 overall $F(3,48)$=168.6, (prob>F)=0.0001			
Variable	Partial R^2	F	prob>F
smextime	0.8615	310.9	0.0001
smioccom	0.0289	12.9	0.0007
CCR1	0.0229	12.7	0.0008

Set R			
total R^2=0.6638 overall $F(3,140)$=92.1, (prob>F)=0.0001			
Variable	Partial R^2	F	prob>F
smextime	0.6483	261.7	0.0001
CCR1	0.0057	2.3	0.1299
smrdcnt	0.0098	4.1	0.0458

Figure 6-26:
Stepwise Regression on t_{diff}
using Four Independent Variables

Set N		
R^2=0.9134		
$F(4,47)$=123.9, (prob>F)=0.0001		
Variable	t	prob>|t|
smextime	11.3	0.0001
CCR1	-3.5	0.0009
smioccom	-1.6	0.1118
smrdcnt	0.2	0.8334

Set R		
R^2: 0.6640		
overall $F(4,139)$=68.7, (prob>F)=0.0001		
Variable	t	prob>|t|
CCR1	2.1	0.0334
smrdcnt	-1.9	0.0620
smextime	-1.7	0.1013
smioccom	0.3	0.7435

Figure 6-27:
Regular Regression on t_{diff}
using Four Independent Variables

In set R, the stepwise regression using four variables has exactly the same results as the stepwise regression using 17 variables.

The R^2 values are slightly lower with four variables than with 17. Some of the modelling power, or predictive power, has been lost, but the four variables seem to have most of the power of the set of 17 variables.

6-7.3.5. One Variable

The variable *smextime* seems to be the one variable with the strongest predictive power. To confirm this we ran a regression of t_{diff} using only *smextime*. For set N, R^2=0.8615, $F(1,50)$=310.9, (prob>F)=0.0001, t for *smextime* is 17.6, and (prob>|t|)=0.0001. For set R, R^2=0.6483, $F(1,142)$=261.8, (prob>F)=0.0001, t for *smextime* is 16.2, and (prob>|t|)=0.0001.

It appears that *smrdcnt* is the variable with next most predictive power after *smextime*. We ran a regression with *smextime* and *smrdcnt*, to see how much improvement in R^2 is obtained by adding the second variable. The improvement is negligible, from 0.6483 to 0.6521 in set R.

6-7.3.6. Summary of the Three Sets of Variables

We have seen that the set of four variables seems to capture most of the predictive power of the set of 17 variables, in both sets N and R, as measured by R^2, and that the single variable *smextime* seems to capture most of the predictive power of the two larger sets of variables in both sets N and R. This is summarized in Figure 6-28.

	R^2		
	17 variables	4 variables	1 variable
Set N — stepwise	0.95	0.91	
Set N — regular	0.96	0.91	0.86
Set R — stepwise	0.66	0.66	
Set R — regular	0.74	0.66	0.65

Figure 6-28:
Predictive Power (R^2)
of Three Sets of Independent Variables

For comparison, regular regression of t_{diff} using only *CCR1* gives R^2=0.6382 in set R, which is only slightly worse, while in set N, R^2=0.1012, which is very much worse.

6-7.3.7. Other Analyses

6-7.3.7.1. Other Measures of CPU Requirement

The variable *smextime* is a measure of CPU demand, so we investigated other CPU requirement variables a bit further. All the *CR* variables are very highly correlated, including those related to individual processes and those that represent the means of two processes: *f0cr*, *f1cr*, *fmcr*, *af0crap*, *af1crap*, *afmcrap*, *af0crtog*, *af1crtog*, and *afmcrtog*. Furthermore, all the *CR* variables are very highly correlated with *smextime*. In set R the correlation coefficients range from 0.81 to 1.0, with most between 0.95 and 1.0. In set N the correlation coefficients range from 0.92 to 1.0. Stepwise regression on t_{diff} using the *CR* variables and *smextime* shows that *afmcrtog* is the most important variable in both set N and set R; *afmcrtog* is the mean CPU requirement of the processes in the TOGETHER configuration. Regression of t_{diff} using only *afmcrtog* has R^2 = 0.8632 in set N and R^2 = 0.6488 in set R. These are virtually the same results as for *smextime*, which is not surprising, since the correlation between *afmcrtog* and *smextime* is high.

6-7.3.7.2. CCR3 Alone

We ran a regression on t_{diff} using only *CCR3*. In set R, R^2 = 0.6554. This is similar to the result of regression using *smextime*, where R^2 = 0.6483. In set N, however, the results were very poor, with R^2 = 0.1184. Even the R^2 for t_{APART} was only 0.3948, which is unusual (in almost every other regression $R^2 > 0.9700$ for t_{APART} and t_{TOG}). The poor results in set N may be partly due to the zero values assigned when *smrcvcnt* = 0. The low R^2 of *CCR1* in set N, mentioned above, could be due to the same cause; we will further examine this possibility below.

6-7.3.7.3. The CCR Plus One Variables

CCR1, *CCR2*, and *CCR3* are various ratios of computation to communication. The value of a CCR is set to zero if the denominator is zero, so a CCR could have the value zero either because its numerator is zero or because the denominator is zero.

We created three new ratios similar to the old CCR ratios but with the value one added to the denominators. For example, $CCR1 = \frac{CR+I}{rcvCount}$, the new ratio is computed as $\frac{CR+I}{rcvCount+1}$. Since the old denominators are measures of the amount of communication, the new denominators give the number of inter-communication intervals. The new ratios are named *CCR1p1*, *CCR2p1*, and *CCR3p1* (the ‘‘*p1*’’ stands for ‘‘plus one’’). All of these variables have values different from the old CCRs, in both set N and set R. The results of regressions using the *p1* CCRs are slightly better than previous results, but not much better.

6-7.3.7.4. Percentage of Communication

Another ratio of computation to communication is *smsrpct*, the percentage of actions that are communications, that is, the percentage of all actions that are sends or receives, taken separately for each process, then averaged over both processes. A regression of t_{diff} using *smsrpct* gave, in set R, $R^2 = 0.3638$, and in set N, $R^2 = 0.3959$. Clearly, this measure of the computation/communication ratio is not a good predictor of t_{diff}.

6-7.3.7.5. Waiting Time

We also investigated the amount of time processes spend waiting for messages to arrive. The variable *afmwap* is the mean total amount of time each process spent waiting for messages to arrive in the APART configuration, averaged over the two processes. The variable *afmwtog* is similarly defined for the TOGETHER configuration.

We ran regressions of t_{diff} using *afmwap* and *afmwtog*. All the regressions produced very low values of R^2 in both set N and set R. For example, a regression of t_{diff} using both *afmwap* and *afmwtog* produced R^2=0.0058 in set N and R^2=0.4876 in set R. Clearly, this measure of receive waiting time is not useful in choosing placements.

6-7.3.7.6. Elapsed Time Variables

The actual CPU times (such as *afmcrtog*) and actual I/O times (such as *af0iap*), described in Section 6-7.2.2, are service times. The three-process matching analysis described in Section 7-3.3, below, indicates that *elapsed* CPU time, rather than CPU service time, and elapsed communication waiting time are useful in selecting fast configurations. The elapsed CPU time for a process is the wall clock time the process spends waiting for and obtaining CPU service, the sum of the times elapsed between a CPU service request and the completion of the request. The matching analysis also indicates that the elapsed times from the TOGETHER simulations provide better predictions than the values from the APART simulations.

We extracted from the simulation outputs the actual elapsed CPU and I/O times, and the actual elapsed communication waiting times. There are CPU, I/O, and waiting times for each of the two processes, and we computed the mean per-process values. The actual elapsed communication waiting times are the same as the actual waiting time variables such as *af0wap, afmwtog*, and so on described in Section 6-7.2.2 and shown in Figure 6-18. In addition, we extracted the total elapsed time for each process. The total elapsed time of the distributed computation, the response time, in a particular configuration, is the longest process elapsed time. We only extracted the values from the TOGETHER simulations.

These new variable names begin with ‘‘*atog,*’’ for ‘‘actual elapsed time from the TOGETHER simulation.’’ Following the ‘‘atog’’ is a ‘‘*0,*’’ a ‘‘*1,*’’ or an ‘‘*m,*’’ indicating whether the variable represents a value from process 0, process 1, or the mean of the two processes, respectively. The elapsed CPU time variables have names ending in ‘‘*cpu,*’’ the elapsed I/O time variables have names ending in ‘‘*io,*’’ the communication waiting time variables have names ending in ‘‘*comm,*’’ and the total process elapsed time variables have names ending in ‘‘*tot.*’’ The complete list of *atog* variables is shown in Figure 6-29.

Variable Name for Process 0	Variable Name for Process 1	Variable Name for Mean of Procs.	Description
atog0tot	*atog1tot*	*atogmtot*	actual total elapsed time
atog0cpu	*atog1cpu*	*atogmcpu*	actual elapsed CPU time
atog0io	*atog1io*	*atogmio*	actual elapsed I/O time
atog0com	*atog1com*	*atogmcom*	actual communication waiting time

Figure 6-29:
The *atog* Outcome Variables
from the TOGETHER Simulation Outputs

We extracted these actual elapsed time values from experiment set R and added the variables to data set R.

The results of regressions using these new variables are good. Regressions of t_{diff} using only the new elapsed time variables have R^2 of about 0.95 in set R. Regressions using the new variables in addition to other variables have R^2 of about 0.98. Without the elapsed time variables, the best R^2 for t_{diff} is about 0.68.

In a stepwise regression of t_{diff} using the reduced set of 17 variables plus the new elapsed time variables, *atogmcpu* is the first variable selected, with a partial R^2 of 0.7536. *Smextime* is selected second, with a partial R^2 of 0.1944. The other variables selected all have much smaller partial R^2 of 0.0200 or less.

It is not surprising that the predictions of t_{diff} using the elapsed time variables are much better than the predictions based on service times. The dependent variable t_{diff} is the difference between two elapsed times. Using elapsed times to predict elapsed times obviously works better.

These elapsed times are dependent not only on the characteristics of the distributed program and its input, but also on the characteristics of the computer system used and on the load on the computer system. We would much prefer to be able to predict t_{diff} using only the characteristics of the program, or the program and data. (Actually, all that is needed to choose a configuration is the sign of t_{diff}.) If we could do that, then the characterization of the program and data could be done using any computer system, under any load conditions, not necessarily using the production system or under production conditions. If, however, the elapsed time variables are needed to make a placement decision, then we must run the program under production conditions to obtain those values. Furthermore, and perhaps more importantly, if the production environment changes, for example, if the loads on the CPUs change, then the placement decision might no longer be correct.

It is quite interesting that elapsed time values from executing the program TOGETHER allow us to predict with reasonable accuracy t_{diff}, and hence t_{APART}; it is not necessary to execute the program in every configuration. Also, it may be easier to measure actual elapsed times than service times or process summary variables.

In summary, the elapsed time values are powerful predictors, but they have serious drawbacks; it would be preferable to use only system-independent program characteristics. On the other hand, executing the program TOGETHER provides surprisingly good information about which configuration to use.

6-7.4. The Synchronization Proportionality Factor (SPF)

6-7.4.1. Introduction

This section is a slight digression from our attempts to predict t_{diff}. In it, we discuss how much of processes' communication waiting time is due to communication delay and how much to synchronization.

When a process sends a message, the message takes a certain amount of time to pass through the communication network and arrive at the destination process' incoming message queue; call this the *communication transit time, c*.

When a process performs a receive action, it takes the first available message from its incoming message queue. If there are no messages in its incoming queue, the process blocks until a message arrives. The *wait time, w*, for a receive operation is the elapsed time from the moment the receiving process starts the receive action until the moment the receive action is completed, including any time the process spends blocked. Part of this wait time is the communication transit delay; let us call the remaining part of the wait *synchronization delay, s*, so $w = c + s$. The wait time can be greater than, less than, or equal to the communication delay. If the sender sends at the same moment the receiver blocks, then the wait time is equal to the communication time. If the sender sends later than that, then the wait time is greater than the communication time, and we call the increase a positive synchronization delay. If the sender sends earlier than that moment, then the wait time is less than the communication time, and we call the decrease a negative synchronization delay.

How much of a distributed program's waiting time is due to communication delay, *c*, and how much is due to synchronization delay, *s*? If the total communication delay is larger than the total synchronization delay, then efforts to improve performance can focus on increasing the network speed. If, on the other hand, a large part of the waiting time is due to synchronization, then efforts to improve program performance might focus on tuning the program to avoid long waits.

To compare *c* and *s*, we define the *synchronization proportionality factor*, or *SPF*, as $SPF = s/c$. Note that $w = c + (c \cdot SPF) = c\,(1 + SPF)$ and $SPF = (w/c) - 1$.

For each run in sets N and R we computed the SPF based on the response time of the APART simulation. In the APART configuration there is no competition for the CPU and I/O facilities, so the elapsed CPU and I/O times should be the same as the service times. The total response time, then, is the sum of the CPU time, the I/O time, and the communication waiting time: $t_{APART} = cr + i + w$, so $w = t_{APART} - cr - i$. Since $SPF = (w/c) - 1$, $SPF = [(t_{APART} - cr - i)/c\,] - 1$. We used the actual values of t_{APART}, *cr*, *i*, and *c* to compute the actual *SPF*.

6-7.4.2. Results

In set N there are 12 runs in which $c = 0$; in those cases SPF is not defined. Summary statistics on the remaining runs in set N and on all runs in set R are shown in Figure 6-30.

Set	# Runs in Set	SPF			
		Mean	Std. Dev.	Minimum	Maximum
Set N	40	153.29	470.42	0.43	2,975.33
Set R	144	3.91	2.34	0.29	9.78

Figure 6-30:
Summary Statistics on SPF

In grid cells with much communication, processes wishing to receive a message need not wait long for another process to send a message, so the SPF is relatively smaller. In cells where there is little communication, such as the lower and right-hand parts of the full grid, a process wishing to receive a message may have to wait a long time before another process sends a message, so the SPF is relatively larger. In set N, processes on the average spend 153 times longer waiting for another process to send a message as

they do waiting for the message to arrive.

Though much smaller than in set N, the mean SPF in set R is still large. The amount of communication is very large in every run in set R, yet even so, on average, processes spend nearly four times as much time waiting for another process to send a message as they do waiting for the message to arrive.

Recall that processes are unlikely to do a receive unless some other process has sent or soon will send a message to them.‡ Considering this, the large SPF for set R and the very large SPF for set N are even more remarkable.

6-7.4.3. Conclusions

There are a number of possible conclusions which might be drawn. One is that the best way to improve the performance of the type of distributed programs we are studying is to improve the synchronization, so that processes do not spend so long waiting. One way to do this is to construct programs and system calls so as to avoid blocking receives.

We can also conclude that improving network speed would not greatly improve the performance of these programs, since they spend relatively little time waiting for messages to be transmitted. On the other hand, we could regard synchronization as fixed by the program function, so that there is little opportunity for improvement in s. In that case, improving the network performance, that is, decreasing c, will decrease $w = c + s$, but it will also increase the SPF.

Since response time does not depend very strongly on network speed, our results should apply to metropolitan-area networks (MANs) and wide-area networks (WANs), even if they are much slower than the networks we model, and even if the speed of message transmission is not uniform between all source/destination pairs.

6-8. Using the Results of the Statistical Analyses: PRED1 and PRED4

6-8.1. Introduction: the PRED1 and PRED4 Programs

The statistical analyses described above were done to find out how to choose the fastest configuration, and produced formulas which predict t_{diff} moderately well. In this section we describe placement procedures which select a configuration using those formulas. The configurations selected are then compared to the actual best configurations according to the simulation outcomes, and the results of the comparisons are analyzed.

The statistical analyses showed that the single best variable for predicting t_{diff} is the mean of the sums of the CPU execution times requested by the two processes, *smextime*; *smextime* also makes good predictions of t_{APART} and t_{TOG}. The SAS regression procedure also provides empirical models, formulas for computing the dependent variables from the independent variables. The formulas given by SAS for computing t_{APART}, t_{TOG}, and t_{diff} using *smextime* are:

$$t_{APART} = 1508.05987 + (1.7215 \cdot smextime) \quad (1)$$

$$t_{TOG} = 1429.6605 + (1.9330 \cdot smextime) \quad (2)$$

$$t_{diff} = -78.3993 + (0.2116 \cdot smextime) \quad (3)$$

‡ While a trace is being generated, when random selection of the next action type for a process indicates a communication action, if more sends to the process have been generated than receives by the process, so that in simulation one or more messages are in the process' message queue or are or soon will be on their way to the process, then there is a 70% chance that the next action for the process will be a receive, and only 30% chance that it will be a send. If more receives have already been generated for the process than sends to it, then it is only 30% likely to generate a receive and is 70% likely to generate a send. See Sections 5-3 and 5-4 for further details.

The statistical analyses also showed that the best small group of variables for predicting t_{diff} consists of *smextime*, *CCR3p1*, *smrdcnt*, and *smioccom*. These variables, too, can also be used to make good predictions of t_{APART} and t_{TOG}. The formulas given by SAS are:

$$t_{APART} = 1029.17901 + (-278.68056 \cdot CCR3p1) + (3.23487542 \cdot smextime) + (67.16986404 \cdot smrdcnt) + (46.22137211 \cdot smioccom) \quad (4)$$

$$t_{TOG} = 592.40771 + (-63.57080283 \cdot CCR3p1) + (2.28271030 \cdot smextime) + (66.54039578 \cdot smrdcnt) + (442.69138 \cdot smioccom) \quad (5)$$

$$t_{diff} = -436.77132 + (215.10976 \cdot CCR3p1) + (-0.95216512 \cdot smextime) + (-0.62946822 \cdot smrdcnt) + (396.47001 \cdot smioccom) \quad (6)$$

We wrote two programs to test the accuracy of predictions made using these formulas. One program, called PRED1, uses the single-variable formulas (equations 1, 2, and 3), the other, called PRED4, uses the four-variable formulas (equations 4, 5, and 6). The programs are identical in all respects other than the formulas used to compute the predicted values.

The PRED programs collect the actual values of t_{APART}, t_{TOG}, and t_{diff}, then compute predicted values of the same quantities. The difference between each actual value and the corresponding predicted value is computed, *actual−predicted*; we call this difference the *residual*, and its value is in milliseconds. The PRED programs also compute the relative percentage error of each residual, which is $(residual/actual) \cdot 100$. Thus there are 12 values for each run: actual value, predicted value, the residual, and the relative error of the residual, for each of t_{APART}, t_{TOG}, and t_{diff}. For example, the 12 values collected or computed by PRED1 for run **rar-4** are shown in Figure 6-31.

	Actual	Predicted	Residual	Relative Error
t_{APART}	3,278.19	2,754.81	523.38	15.97%
t_{TOG}	3,332.83	2,829.58	503.24	15.10%
t_{diff}	54.64	74.85	-20.21	-36.99%

Figure 6-31:
PRED1 values for **rar-4**

To get a measure of how good the predictions are over an entire experiment set, we compute the means of the residuals and the relative errors. The actual and predicted response times for TOGETHER and APART are positive numbers of milliseconds. The actual and predicted t_{diff} values may be positive or negative, depending on which configuration is faster. The residuals and relative errors are differences between actual and predicted values, and they, too, may be positive or negative. When taking the mean of any of these differences, some of the positive and negative values will cancel, so that the mean may be close to zero. Indeed, it is our hope that the means of the residuals and the relative errors will be close to zero, for that would mean that the predictions are good. At the same time, such a mean gives little information

about typical values of these differences, so we also compute the means of the absolute values; this gives us the average magnitude of the residuals and relative errors.

6-8.2. Results of Predictions

The means and average magnitudes of the 12 PRED1 values and the 12 PRED4 values over set R are shown in Figure 6-32.

PRED1				
	Actual	Predicted	Residual	Relative Error
mean t_{APART}	3,344.71	3,344.76	-0.05	-1.33%
avg. magnitude t_{APART}	3,344.71	3,344.76	241.93	8.78%
mean t_{TOG}	3,492.03	3,492.01	0.02	-1.24%
avg. magnitude t_{TOG}	3,492.03	3,492.01	241.52	9.06%
mean t_{diff}	147.33	147.36	-0.03	-101.17%
avg. magnitude t_{diff}	172.65	174.76	91.59	430.36%

PRED4				
	Actual	Predicted	Residual	Relative Error
mean t_{APART}	3,344.71	3,344.71	-0.00†	-0.09%
avg. magnitude t_{APART}	3,344.71	3,344.71	91.28	2.80%
mean t_{TOG}	3,492.03	3,492.03	-0.00††	0.02%
avg. magnitude t_{TOG}	3,492.03	3,492.03	38.22	1.23%
mean t_{diff}	147.33	147.33	0.00†††	-74.92%
avg. magnitude t_{diff}	172.65	166.80	88.19	361.90%

†: -0.000009
††: -0.000004
†††: 0.000007

Figure 6-32:
Means and Average Magnitudes of PRED values
for PRED1 and PRED4 over Set R

The relative errors of the residual for t_{APART} and t_{TOG} are quite small (for PRED1 slightly larger than 1%, for PRED4 smaller than 0.1%), which means that the formulas for predicting the APART and TOGETHER response times are very successful on average. The average magnitude of the relative errors is also fairly small (under 10% for PRED1, 1%-3% for PRED4), showing that the formulas are not only successful on average, but that the accuracy for individual runs is good. It appears that PRED4 is significantly better than PRED1 in that the mean relative error and the average magnitude of the relative error are both much smaller for PRED4 than for PRED1.

The means of the t_{diff} residuals are small (in the case of PRED4, very small), indicating that the formulas for computing t_{diff} (equations 3 and 6) are, on average, quite good. However, the average magnitudes of the residuals are much larger. This would seem to indicate that the individual residuals are larger numbers, but that some of them are positive and some negative, so that the mean is near zero. This is confirmed by the large mean relative errors, and even more by the average magnitudes of the relative errors, which are several hundred percent. We conclude that, although the formulas for t_{diff} are good on average, they are highly inaccurate in individual cases. For t_{diff}, as for t_{APART} and t_{TOG}, the results from PRED4 are significantly better than the results from PRED1, though both sets of results are disappointing.

The relative errors of t_{diff} are much worse than we expected. We computed the standard deviations of the relative errors and the absolute values of the relative errors; the results are shown in Figure 6-33.

PRED1		
	Mean	Standard Deviation
Relative Error	-101.17%	2,258.75%
|Relative Error|	430.36%	2,219.68%

PRED4		
	Mean	Standard Deviation
Relative Error	-74.92%	1,850.87%
|Relative Error|	361.90%	1,816.69%

Figure 6-33:
Means and standard deviations
of the mean and average magnitude
of the t_{diff} relative error
(set R; N = 144)

These standard deviations are very large, which may indicate that the large relative errors might be due to a few very large errors, with the rest of the errors being of more moderate size. To see if this is the case, we examined the distributions of the absolute values of the relative errors, which are shown in Figure 6-34.

PRED1		
Range		Number
From	To	in Range
0.00%	999.99%	134
1,000.00%	1,999.99%	5
2,000.00%	2,999.99%	2
3,000.00%	3,999.99%	1
4,000.00%	4,999.99%	1
...	...	...
25,000.00%	25,999.99%	1

PRED4		
Range		Number
From	To	in Range
0.00%	999.99%	135
1,000.00%	1,999.99%	4
2,000.00%	2,999.99%	3
3,000.00%	3,999.99%	1
4,000.00%	4,999.99%	0
...	...	...
21,000.00%	21,999.99%	1

Figure 6-34:
Distribution of Magnitude of Relative Percentage Errors
(set R; N = 144)

There are indeed a few very large relative errors in the thousands of percents, and one extremely large relative error in the tens of thousands of percents, which would account for the large means and average magnitudes.

The one extremely large relative error occurs in case **ran-3** with both PRED1 and PRED4. These extremely large values are due to the combination of large predicted values of t_{diff} and very small actual

values of t_{diff}. The relative error is computed as $(residual/actual) \cdot 100$; in **ran-3** the residual is fairly large, and dividing by the very small actual t_{diff} produces the extremely large result. Naturally, the huge relative error of **ran-3** has a strong effect on the overall results. To see how strong the effect is, we computed the means and standard deviations of the relative errors of t_{diff} excluding **ran-3**. The results are shown in Figure 6-35.

PRED1		
	Mean	Standard Deviation
Relative Error	79.47%	662.71%
|Relative Error|	252.03%	618.04%

PRED4		
	Mean	Standard Deviation
Relative Error	72.99%	547.09%
|Relative Error|	215.99%	507.92%

Figure 6-35:
Means and standard deviations
of the mean and average magnitude
of the t_{diff} relative error,
excluding case **ran-3**.
(N = 143)

Excluding case **ran-3** gives us some improvement in the mean size of the relative errors, and it gives us a large improvement in the average magnitude of the relative errors. Nonetheless, it appears that the formulas are not very good for making individual predictions of t_{diff}.

In the next section we will use the predicted values for t_{diff} to predict which configuration is the faster, and see how well the predicted configurations match the actual faster configurations.

6-8.3. Hits and Misses

Since t_{diff} is computed as $t_{TOG} - t_{APART}$, we can use the sign of t_{diff} to determine which configuration to use. If t_{diff} is positive, t_{TOG} is larger than t_{APART}, so APART is the faster configuration. If t_{diff} is negative, t_{APART} is larger than t_{TOG}, so TOGETHER is the faster configuration.

The configuration selected by the sign of the predicted t_{diff}, as computed by formula 3 or formula 6, is compared to the configuration indicated by the actual t_{diff}, yielding one of the four possible outcomes shown in Figure 6-36.

Predicted	Actual	
	APART	TOGETHER
APART	hit-a/a	miss-a/t
TOGETHER	miss-t/a	hit-t/t

Figure 6-36:
Possible Hit or Miss Outcomes of PRED

A ‘‘hit’’ means that the prediction correctly ‘‘hit’’ the actual faster configuration, a ‘‘miss’’ means that it did not. The notations ‘‘a/a,’’ ‘‘a/t,’’ ‘‘t/a,’’ and ‘‘t/t’’ abbreviate the predicted and actual best configurations, respectively. For example, ‘‘miss-t/a’’ means that the prediction missed predicting the actual best placement, the prediction being TOGETHER and the actual best configuration being APART.

PRED1 and PRED4 each compute a hit-or-miss outcome for each run in set R. Figure 6-37 shows the number of times each of the hit-or-miss outcomes occurs and the percentage of runs in set R in which

each outcome occurs.

PRED1		
	# Runs	% of Runs
TOGETHER/TOGETHER hits	20	13.89%
APART/APART hits	97	67.36%
Subtotal of hits	117	81.25%
TOGETHER/APART misses	14	9.72%
APART/TOGETHER misses	13	9.03%
Subtotal of misses	27	18.75%
Total	144	100.00%

PRED4		
	# Runs	% of Runs
TOGETHER/TOGETHER hits	19	13.19%
APART/APART hits	99	68.75%
Subtotal of hits	118	81.94%
TOGETHER/APART misses	12	8.33%
APART/TOGETHER misses	14	9.72%
Subtotal of misses	26	18.06%
Total	144	100.00%

Figure 6-37:
Hit and Miss Rates for PRED1 and PRED4

Both PRED1 and PRED4 are quite successful with 81-82% hits. Most of those hits, 67-69%, are APART/APART hits. The 81% success rate is greater than we had been led to expect by the R^2 values of 0.64 to 0.68 in the regressions of t_{diff} (see Section 6-7.3), and by the large average magnitude of the relative errors in predicting t_{diff} (see Section 6-8.2).

We continue the analysis of the PRED placement techniques by more closely examining the misses. Where in the grid do the misses occur? Are they TOGETHER/APART or APART/TOGETHER misses? The PRED1 hit-or-miss outcome for each run in set R is shown in Figure 6-38.

	IO-COMM					
	IO COMM	IO COMM	IO COMM	IO COMM	IO COMM	IO COMM
CPU	0 100	1 99	2 98	3 97	4 96	5 95
0	**raa**	**rab**	**rac**	**rad**	**rae**	**raf**
	hit-t/t	miss-t/a	hit-t/t	hit-t/t	miss-t/a	hit-t/t
	hit-t/t	hit-t/t	hit-t/t	miss-t/a	hit-t/t	hit-t/t
	hit-t/t	hit-t/t	hit-t/t	miss-t/a	hit-t/t	miss-t/a
	hit-t/t	hit-t/t	hit-t/t	hit-t/t	miss-t/a	miss-t/a
1	**rag**	**rah**	**rai**	**raj**	**rak**	**ral**
	hit-a/a	hit-a/a	hit-a/a	miss-t/a	miss-t/a	hit-a/a
	hit-t/t	hit-a/a	miss-a/t	miss-a/t	miss-a/t	miss-t/a
	miss-a/t	miss-a/t	hit-a/a	hit-t/t	miss-t/a	miss-t/a
	hit-a/a	hit-t/t	miss-t/a	miss-t/a	hit-a/a	miss-a/t
2	**ram**	**ran**	**rao**	**rap**	**raq**	**rar**
	hit-a/a	miss-a/t	hit-a/a	hit-a/a	hit-a/a	hit-a/a
	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a
	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a
	miss-a/t	hit-a/a	miss-a/t	hit-a/a	hit-a/a	hit-a/a
3	**ras**	**rat**	**rau**	**rav**	**raw**	**rax**
	hit-a/a	miss-a/t	hit-a/a	hit-a/a	hit-a/a	hit-a/a
	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a
	hit-a/a	hit-a/a	hit-a/a	hit-a/a	miss-a/t	hit-a/a
	miss-a/t	hit-a/a	miss-a/t	hit-a/a	hit-a/a	hit-a/a
4	**ray**	**raz**	**rba**	**rbb**	**rbc**	**rbd**
	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a
	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a
	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a
	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a
5	**rbe**	**rbf**	**rbg**	**rbh**	**rbi**	**rbj**
	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a
	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a
	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a
	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a	hit-a/a

Figure 6-38:
PRED1 Hit or Miss Outcomes in Set R

The PRED4 outcomes are different from the PRED1 outcomes in only three cases; these are shown in Figure 6-39.

Run	PRED1	PRED4
rag-2	hit-t/t	miss-a/t
rai-4	miss-t/a	hit-a/a
ral-3	miss-t/a	hit-a/a

Figure 6-39:
Differences Between PRED1 and PRED4

Notice that, in the first row, all predictions by both PRED1 and PRED4 are TOGETHER, in the second row the predictions are mixed, and in the third row and below all predictions are APART. The three differences between PRED1 and PRED4 are all in the second row.

For both PRED1 and PRED4, there seems to be some concentration of misses in the upper right corner of the zoomed-in grid. Most of the misses occur in the top two rows of the grid, and all of the misses occur in the top four rows, those with 3% or less CPU actions. There are no misses in the last two rows and we presume that there would be no misses below the bottom of the zoomed-in grid. We

presume that very few real programs fall into the top four rows of the grid, so we could use PRED1 or PRED4 (probably PRED1 for simplicity), knowing that it always makes the best placement when operating below the fourth row but that it sometimes misses when the percentage of CPU actions is very small. However, the transition zone is entirely within the first four rows of the grid. If we are to assume that real distributed programs almost never fall into the first four rows, we can achieve near-perfect success by using the always-APART policy. Always-APART would be far simpler to implement and faster to execute than PRED1 or PRED4.

We have discussed hits and misses on a per run basis, but there are four runs per grid cell in set R; how good are the PRED1 and PRED4 predictions on a per cell basis? Figure 6-40 shows how frequently there are 1, 2, 3, or 4 misses per cell for each PRED program.

# Misses	PRED1	PRED4
1	10	10
2	4	5
3	3	2
4	0	0

Figure 6-40:
Number of Misses per Cell in Set R

There are never four misses per cell.

Let us say that PRED has failed in a grid cell if there are two or more misses in the cell. By this definition, each of the PRED programs has seven failures in the zoomed-in grid, which is 19.44% of the 36 cells. This leaves 80.56% of the cells in which PRED succeeds, which is very close to the overall percentage of successful runs. If we consider even one miss in a cell to be a failure, then each of the PRED techniques has 17 failures out of 36 cells, or 47.22% failure, leaving 52.78% success cells, which is not bad considering the strictness of the criteria.

Both PRED techniques miss only in the upper rows of the grid. The size and shape of the zoomed-in grid is arbitrary; if the grid were several rows larger, there would probably be no more misses, and the success rate would be higher. On the other hand, if the grid were wider, the cells further to the right in the first two rows might have a number of misses. Is the 6-by-6 zoomed-in grid a good size and shape?

The zoomed-in grid is supposed to allow examination of the upper left corner. A square grid seems a reasonable shape for a corner. Let us examine the 5-by-5 sub-grid from **raa** to **rbc**. Again counting two or more misses as a failure, there are five failures out of 25 cells, or 20.00% failures, which is 80.00% successes. Now let us imagine increasing the grid to 7-by-7. Most likely, the top two cells of the new column, immediately to the right of **raf** and **ral**, would be failures, and all other new cells would be successes. That would give us nine failures out of 49 cells, or 18.37% failures, which is 81.63% successes. Both of these success rates are quite close to the 80.56% success rate of the 6-by-6 grid. Thus, 80-81% success for the PRED techniques in the upper left corner seems to be accurate, and not dependent on the size of the zoomed-in grid.

6-8.4. Summary

We have seen that formulas (1), (2), (4), and (5) for computing t_{APART} and t_{TOG} are reasonably accurate in set R, the one-variable formulas being within 10% and the four-variable formulas being within 3%. Unfortunately, formulas (3) and (6) for t_{diff} are not very accurate, the mean relative percentage error in the magnitude of t_{diff} being 430% for one variable and 362% for four variables. Part of these large mean relative errors are due to eight or nine very large relative errors and one huge relative error. However, without the huge value, the mean relative percentage errors in the magnitude of t_{diff} are 252% for one variable and 216% for four variables, still quite large.

We used the sign of the predicted t_{diff} to predict which configuration would be faster, then compared the predicted configuration to the actual faster configuration. The result was about 81-82% successful predictions, or ‘‘hits,’’ in set R, with both one variable and four variables. The one-variable prediction and the four-variable prediction are different in only three runs out of 144 in set R, so apparently the additional three variables add very little. All the differences are in the 1% CPU row.

All the misses, with both one variable and four, are in the top four rows of the zoomed-in grid, those with 3% or less CPU execute actions. If we assume that real programs almost never fall into the top four rows, either the one-variable or the four-variable prediction would be almost perfectly accurate. On the other hand, all the TOGETHER wins are within the top four rows; so, if we are to make the assumption that real programs never fall into the top rows of the grid, we should just use the always-APART policy.

In a practical sense, the one-variable formulas seem to be as useful as the four-variable formulas. The one variable used is *smextime*, the mean CPU service time requested by the processes. Practically, then, the only characteristic that matters in making placement decisions is the CPU time demanded.

6-9. Experiment Set S

6-9.1. Introduction

In experiment sets N and R, the number of actions in each process is fixed using the termSync feature of GENACTIONS (see Section 5-3.6). Using the termSync feature might possibly have distorted the generated traces in some unanticipated way. To address this concern, we created experiment set S. The traces in set S were generated with the termSync feature turned off, as opposed to sets N and R and all our other experiment sets, which did use the termSync feature. This allows us to test for artifacts of the termSync feature. It also produces widely varied trace lengths, which we can use to test for any effects of same-length traces.

6-9.2. Traces

In experiment sets N and R the termSync feature had been set to generate traces of about 800 actions, or about 400 actions for each of the two processes. For set S, we set the probability of a terminate action being generated for a process to $1/400 = 0.0025$. In previous experiment sets, the probability of generating a terminate action was set to zero, and terminates were generated through the termSync feature. In set S, we ‘‘steal’’ the probability for a terminate action from the probability of communication, since in set R all runs have nonzero probability of communication, whereas some runs have zero probability of execution or I/O actions. Specifically, we steal the probability for a terminate action from whichever of prob_send or prob_rcv is larger. This is prob_send in the ‘‘regular’’ state and prob_rcv in the ‘‘message-waiting’’ state (see Sections 5-3 and 5-4). In order to ensure reasonably long runs, any trace that contained a process of less than 300 actions was discarded and a new trace generated; 300 was chosen arbitrarily. Other than these changes, set S is a repeat of some of the runs from set R.

Rather than repeating the entire set R grid, we repeat only the grid cells in the transition zone. Specifically, we repeat runs **raj**, **rak**, **ral**, **rap**, **raq**, **rar**, **ras**, **rat**, **rau**, **ray**, **raz**, and **rba**. As in set R, there are four repetitions per cell.

Figure 6-41 shows summary statistics on the number of actions per process in each trace for experiment sets N, R, and S.

Set	# Procs. in Set	# Actions per Process			
		Mean	Std. Dev.	Minimum	Maximum
Set N	104	402.18	28.64	260	543
Set R	288	401.42	3.50	391	411
Set S	96	531.42	167.42	325	949

Figure 6-41:
Summary Statistics
on the Number of Actions per Process in each Trace

The average number of actions per process in set S turned out somewhat larger than desired, and the longest process is quite long. This is probably due to our discarding traces with less than 300 actions per process.

The large standard deviation for set S led us to wonder about different process lengths in the same trace.† We computed the difference between the numbers of actions for the two processes of each trace in sets N, R, and S; summary statistics are shown in Figure 6-42.

Set	# Runs	Diffs. Between # of Actions per Process			
		Mean	Std. Dev.	Minimum	Maximum
Set N	52	32.63	46.14	0	283
Set R	144	5.30	4.37	0	20
Set S	48	17.42	17.77	0	75

Figure 6-42:
Summary Statistics
on the Difference Between the Number of Actions per Process
in each Trace

The differences in set S are more than three times larger than those for set R, to which it is similar.

Thus, there are three differences between set S and set R. First, the mean number of actions per process is larger in set S. Second, the difference between the number of actions for each of the two processes per trace is much larger in set S. Third, there are fewer cases and fewer runs in set S, and the set S runs are all in two rows of the six-row zoomed-in grid. Any or all of these differences may cause differences in the outcomes in the two sets.

6-9.3. Winning Configurations

The winning configurations in each run in set S are shown in Figure 6-43.

† The GENACTIONS program generates a single trace, which may contain actions for any number of processes. In sets N, R, S, and T, of course, all traces contain actions for two processes. A parameter, *min_actions*, controls the number of actions in the entire trace, not the number of actions for any one process. Thus, when we say that there were approximately 400 actions per process, that means that we set *min_actions* to 800. The process to which each action applies is generated randomly within GENACTIONS, so the 800 actions might be unevenly distributed between the processes. We inspected the traces before simulating them and discarded any which looked too uneven to casual inspection. See Section 5-3.6 for further explanation.

	IO-COMM											
CPU	IO 0	COMM 100	IO 1	COMM 99	IO 2	COMM 98	IO 3	COMM 97	IO 4	COMM 96	IO 5	COMM 95
0												
1							**saj** T A T A		**sak** A A A A		**sal** A A A A	
2							**sap** A A A A		**saq** A A T A		**sar** A A A A	
3	**sas** A A A A		**sat** A T A A		**sau** A A A A							
4	**say** A A A A		**saz** A A A A		**sba** A A A A							
5												

Figure 6-43:
Winning Configurations in Set S

These outcomes are quite similar to the outcomes in the equivalent runs in set R (see Figure 6-4). In seven of the 12 cases, the outcomes are the same in both sets, for example, **raj** and **saj** each have two APART wins and two TOGETHER wins. In the remaining five cases, only one of the outcomes is different, for example, **ras** has three APART wins while **sas** has four.

6-9.4. Statistical Analysis

After running the simulations for set S, we checked to see whether the 17 variables in the reduced set used in set R still have low correlations between them in set S. We did not check whether some variables which had been excluded from the reduced set in set R because they had high correlations with other variables might be included because they have low correlations in set S. There were a number of differences in the correlations.

In set R, *s0sndcnt*, *s0sndsiz*, *s0rcvcnt*, *s1sndcnt*, *s1sndsiz*, and *s1rcvcnt* could not be eliminated because their correlations with *smsndcnt* were low (approximately 0.5); in set S, these variables all have correlations with *smsndcnt* greater than 0.9. In set R, *CCR1* has a correlation of 0.97 with *smextime*; in set S, the correlations between *smextime*, *CCR1*, and *CCR3* are approximately 0.71-0.77, and these are the highest correlations to *smextime* in set S. The correlation between *smrdcnt* and *smwrcnt* is higher in set S at 0.94 than in set R at 0.86. In set R, there is a fairly high correlation of 0.82 between *smsndcnt* and *CCR1*; in set S, the correlation is only 0.05-0.08. The correlation between *smsndcnt* and *smextime* is 0.67 in set R, 0.53 in set S. In set R, there is a low correlation of 0.58 between *smsndcnt* and *smsndsiz*; in set S it is greater than 0.99. Both *smactcnt* and *smterpct* are highly correlated to a number of other variables. *Smiocio* and *smioccom* are not perfectly correlated in set S, as they are in set R, and both are more highly correlated to other variables than they are in set R.

As a result of analyzing the correlations, we eliminated *s0sndcnt*, *s0sndsiz*, *s0rcvcnt*, *s1sndcnt*, *s1sndsiz*, *s1rcvcnt*, *smsndsiz*, *smactcnt*, and *smterpct*. We also eliminated *CCR1inv*.

We added three variables: *CCR1p1*, *CCR3*, and *CCR3p1*. These variables were created after we formed the reduced variable set for experiment set R. The resulting reduced variable set for experiment set S is: *CCR1*, *CCR1p1*, *CCR3*, *CCR3p1*, *smextime*, *smrdcnt*, *smwrcnt*, *smsndcnt*, *smioccom*, and *smiocio* (the reduced set of 17 variables for set R is shown in Figure 6-22).

Having decided on a set of variables to analyze, we analyzed them. Stepwise regression on t_{diff} using the reduced variable set gives a weak R^2 of only 0.2490, $F(1,46)$=15.3 and (prob>F)=0.0003;

smextime is the only important variable, so of course its partial R^2 is 0.2490, F=15.3, and (prob>F)=0.0003. Regular regression on the same variables gives an R^2 of 0.4460, which is better than the stepwise regression, but still worse than the R^2 in set R; $F(10,37)$=3.0 and (prob>F)=0.0075. The important variables are *smioccom* (t=2.0, (prob>$|t|$)=0.0481) and *smiocio* (t=2.0, (prob>$|t|$)=0.0500); *smextime* is not significant (t=1.4, (prob>$|t|$)=0.1723).

Regular regression on t_{diff} using *smextime*, *smrdcnt*, and *smsndcnt* gives weak results: R^2=0.2902, $F(3,44)$=6.0 and (prob>F)=0.0016, and only *smextime* is significant (t=2.2, (prob>$|t|$)=0.0369). Regular regression using only *smextime* gives even weaker results: R^2 is 0.2490, $F(1,46)$=15.3 and (prob>F)=0.0003, and of course *smextime* is significant (t=3.9, (prob>$|t|$)=0.0003).

These low values of R^2 are very disturbing. In set R, R^2 values of about 0.65 let us correctly predict the faster configuration in about 80% of runs (see Section 6-8.3 and Figure 6-37). With the set S R^2 values below 0.45 we will get a much smaller percentage of correct predictions, perhaps 60% less, which would not be very good.

6-9.5. Set S Compared to a Subset of Set R

Set S consists of only ‘‘hard’’ cases, grid cells which are in the transition zone. Part of the success in set R is due to a number of grid cells which are ‘‘easy’’ to predict, particularly the bottom rows and the lower right corner area. Does this mean that for the same hard cases in set R our predictions are only about 60% accurate? If so, that is not very good. On the other hand, if our technique is always right for ‘‘easy’’ cases, and 60% right for ‘‘hard’’ cases, that is not bad at all.

To test whether the poor results from set S are due to the fact that set S includes only hard cases, we created a subset of set R consisting of just those cases which also appear in set S, specifically, **raj**, **rak**, **ral**, **rap**, **raq**, **rar**, **ras**, **rat**, **rau**, **ray**, **raz**, and **rba**. On this subset of set R, we repeated the regression on three variables and regression on one variable previously performed on set S.

The result of regular regression of t_{diff} using *smextime*, *smrdcnt*, and *smsndcnt* was $R^2=0.4192$, weaker than the usual R^2 of 0.65 in set R, but stronger than set S’s R^2 of 0.2902; $F(3,44)$=10.6 and (prob>F)=0.0001. As in set S, only *smextime* is significant (t=3.9, (prob>$|t|$)=0.0003).

The result of regular regression of t_{diff} using only *smextime* was $R^2=0.4039$, again, weaker than the usual R^2 of 0.65 in set R, but stronger than set S’s R^2 of 0.2490; $F(1,46)$=31.2 and (prob>F)=0.0001. Again, *smextime* is significant (t=5.6, (prob>$|t|$)=0.0001).

6-9.6. Conclusions

Apparently, we cannot make our predictions for set S as good as for set R. If real world programs are more like set S, our success with real programs might be significantly less than the 80% we achieved with set R.

6-10. Experiment Set T

6-10.1. Introduction

In experiment set T we investigate widely varying trace lengths to see whether the length of the traces used in sets N and R is adequate. We want to be sure that the simulations are ‘‘long enough,’’ but no longer than necessary.

Simulations of computer systems are often carried out for the purpose of modelling the behaviour of the system, but that is not the purpose of our simulations. Those carrying out system-modelling simulations must be aware that such systems behave differently during some initial start-up period than after they have reached their equilibrium or stable operating state. In system modelling, then, one must determine

the length of the initial period, and usually the initial period is discarded and only data from the stable state is used. We, however, are not modelling the continuing operation of a computer system; rather, we are modelling the execution of particular processes. The most important outcome of the simulations, for us, is the relative response times of the configurations, that is, which configuration is faster. If simulation shows the same configuration is faster when the trace is 200 actions per process as when the trace is 400 actions per process, we can in the future use traces of 200 actions per process. Using shorter traces saves computer time in running the simulations and disk space for storing the traces.

To create experiment set T we chose one grid cell, and, using the characteristics of that cell, generated a number of traces of widely varying lengths. We choose cell **rao** (2% executes; 2%/98% comm/IO), because it is near the center of the zoomed-in grid, and because it is in the transition zone. We generated traces with approximately 100, 200, 400, 600, 800, and 1000 actions per process; these six cases are called **tao100**, **tao200**, **tao400**, **tao600**, **tao800**, and **tao1000**. As in experiment sets R and S, we generated four traces per case, for a total of 24 runs in set T.

6-10.2. Traces

The different length traces were generated by changing the *min_actions* parameter, which is part of the termSync feature of the GENACTIONS program (see Section 5-3.6). Except for the trace length, controlled by the *min_actions* parameter, all other parameters to GENACTIONS were identical to each other and to run **rao**. Recall, however, that GENACTIONS has a granularity of 2-3% (see Section 4-6), so that, even though the input parameters to GENACTIONS are the same except for *min_actions*, the traces generated will have somewhat different characteristics. Since all traces in set R have approximately 400 actions per trace, case **tao400** should be approximately the same as case **rao**.

Figure 6-44 contains two tables with information about the set T traces. The first shows summary statistics on the number of actions per process; the second shows summary statistics on the differences between the number of actions in each of the two processes in each trace.

Case	# of Actions per Process				
	Nominal	Mean	Std. Dev.	Minimum	Maximum
tao100	100	101.50	1.32	100	104
tao200	200	201.63	3.39	195	207
tao400	400	401.38	3.50	396	407
tao600	600	601.13	1.36	599	603
tao800	800	801.13	5.25	791	811
tao1000	1,000	1,001.75	5.07	992	1,010

Case	# Runs	Diffs. Between # of Actions per Process			
		Mean	Std. Dev.	Minimum	Maximum
t100	4	2.00	1.22	0	3
t200	4	5.25	4.15	1	12
t400	4	5.75	3.96	1	11
t600	4	2.25	1.48	0	4
t800	4	7.25	7.60	1	20
t1000	4	8.00	6.00	2	18

Figure 6-44:
Summary Statistics on
Number of Actions per Process
in Set T

The traces have very close to the desired number of actions per process; the termSync feature works well.

The differences between the number of actions per process in each trace are generally similar to the differences in set R (see Figure 6-42).

6-10.3. Winning Configurations

The winning configuration in each run in set T is shown in Figure 6-45.

Case	Winning Configs.
tao100	A A T T
tao200	A A A T
tao400	A A A A
tao600	T A A A
tao800	A A A A
tao1000	T A A A

Figure 6-45:
Winning Configurations in Set T

In case **rao** there are three APART wins and one TOGETHER win.

In all set T cases except **tao100**, APART is the winner either three or four out of four times. Since these outcomes are interspersed, we will treat them as being the same. The conclusion from examining the list of winning configurations is that 100 actions per process is too short in that it produces results different from the results of longer traces, but that traces with 200 actions per process or longer produce the same results.

6-10.4. Response Times

6-10.4.1. Mean Response Times

Figure 6-46 shows the mean response times for each case in set T.

Case	# Runs	Mean Resp. Time (ms)	
		APART	TOGETHER
tao100	4	637	683
tao200	4	1,143	1,190
tao400	4	3,213	3,268
tao600	4	4,268	4,358
tao800	4	5,186	5,263
tao1000	4	7,502	7,640

Figure 6-46:
Mean Response Times in Set T

The mean APART response time of the four runs in case **rao** is 2,761 ms; the mean TOGETHER response time is 2,826 ms. These figures are about 14% smaller than the corresponding response times for **tao400** (the relative percentage differences, relative to the **tao400** times, are 14.07% for APART and 13.53% for TOGETHER).

6-10.4.2. Linear Increase in Response Times

If the increase in response time is not linear with respect to increasing trace length, then we must consider the relationship between trace length and response time in choosing the length of traces. In this section we check whether the relationship is linear.

We compute predicted mean APART and TOGETHER times for cases **tao200** through **tao1000**, based on the prediction that response time does increase linearly with trace length. Taking the mean APART and TOGETHER response times of **tao100** shown in Figure 6-46 as the unit times, the predicted times are computed by multiplying the unit times by two to get the predicted times for **tao200**, by four for **tao400**, and so on. We then compute (1) the difference between the actual mean response time, as shown in Figure 6-46, and the predicted response time, and (2) the relative percentage error of the prediction, computed as $[(actual - predicted)/actual] \cdot 100$. The results of these computations are shown in Figure 6-47.

Case	APART			TOGETHER		
	Predicted	Difference	Rel. Pct. Error	Predicted	Difference	Rel. Pct. Error
tao100	637	0	0	683	0	0
tao200	1,274	-131	-11.46	1,366	-176	-14.79
tao400	2,548	665	20.70	2,732	536	16.40
tao600	3,822	446	10.45	4,098	260	5.97
tao800	5,096	90	1.74	5,464	-201	-3.82
tao1000	6,370	1,132	15.09	6,830	810	10.60

Figure 6-47:
Predicted Linear Response Times in Set T

The predicted times are all within about 20% of the actual times. There is clearly no pattern of increasing or decreasing errors, so we conclude that the response times do increase approximately linearly with respect to the trace lengths. From this point of view, any of the trace lengths in set T would be acceptable.

6-10.5. Response Time Differences

To examine the winning configurations, as we did above, is essentially to examine the difference between the response times of the two configurations, t_{diff}, but to consider only the sign of the difference. In this section we examine the differences in greater detail.

As in previous sections, we compute the absolute differences in milliseconds and the relative percentage differences, and for each we compute summary statistics on the actual values and on the magnitude of the values.

Figure 6-48 shows the summary statistics for each case in set T, as well as the same information for case **rao**.

Case	Absolute Differences (ms)							
	Actual Value				Magnitude			
	Mean	Std. Dev.	Min.	Max	Mean	Std. Dev.	Min.	Max
tao100	46.09	54.30	-9.91	103.96	54.22	46.18	6.35	103.96
tao200	46.98	99.12	-40.73	215.26	67.35	86.58	2.30	215.26
tao400	55.75	31.03	18.96	104.82	55.75	31.03	18.96	104.82
tao600	89.42	106.42	-15.02	264.99	96.93	99.63	15.02	264.99
tao800	77.37	76.25	7.42	202.44	77.37	76.25	7.42	202.44
tao1000	138.34	148.07	-1.27	359.78	138.97	147.47	1.27	359.78

Case	Relative Percentage Differences							
	Actual Value				Magnitude			
	Mean	Std. Dev.	Min.	Max	Mean	Std. Dev.	Min.	Max
tao100	5.81	7.74	-2.03	16.61	7.34	6.32	1.03	16.61
tao200	2.50	6.64	-4.64	13.36	4.82	5.21	0.18	13.36
tao400	1.63	0.74	0.72	2.77	1.63	0.74	0.72	2.77
tao600	1.99	2.27	-0.33	5.69	2.16	2.12	0.33	5.69
tao800	1.40	1.35	0.17	3.60	1.40	1.35	0.17	3.60
tao1000	1.86	2.02	-0.02	4.93	1.87	2.01	0.02	4.93

Differences in Case **rao**								
	Actual Value				Magnitude			
	Mean	Std. Dev.	Min.	Max	Mean	Std. Dev.	Min.	Max
Abs. Diff. (ms)	65.96	72.90	-14.15	180.21	73.04	65.81	14.15	180.21
Rel. Pct. Diff.	2.17	2.41	-0.57	5.88	2.46	2.11	0.57	5.88

Figure 6-48:
Differences in Response Times in Set T
and in Case **rao**

The mean relative percentage differences and the average magnitude relative percentage differences are larger for **tao100** and **tao200**, For **tao400** through **tao1000**, however, the values are all similar, approximately in the range 1.4-2.2%. It appears, then, that within the range of trace lengths studied in set T, any trace length of 400 actions per process or greater has similar results.

6-10.6. Conclusions

We have seen that, within the range of trace lengths studied in set T, response times increase approximately linearly with respect to trace length.

The mean and the average magnitude of the relative percentage differences between the response times of the two configurations are larger for trace lengths of 100 and 200 actions per process; however, these measures have similar values for all trace lengths of 400 actions per process or longer.

To minimize simulation time and disk usage, we want to use the smallest acceptable traces. Our conclusion from studying set T is that 400 actions per process is the shortest acceptable trace length.

6-11. Summary of Two-Process Work

6-11.1. Arrangements and Configurations

In this chapter, we have presented the two configurations possible when there are two processes to be executed and two identical processors available, a 2x2 situation: APART, in which each process is on a separate processor, and TOGETHER, in which both processes are on one processor.

6-11.2. Experiment Sets N, P, and R

We described experiment sets N, P, and R, each of which has two processes per trace. Set N is based on the full grid, which covers the entire space of possible process characteristics. Set P zooms in on the high-communication part of that grid, and set R zooms in even further. In sets N and P, APART wins (has the faster response time) in all but one cell in the upper left corner, the cell with the least CPU use and most communication. Set R, based on the zoomed-in grid, zooms in to the upper left corner far enough that TOGETHER wins in a number of runs.

We found that TOGETHER only wins in runs where 3% or fewer of the actions in the trace are CPU execution actions.

We hypothesize that most real distributed programs will be scientific, computation-intensive programs which do more communication than I/O, and so fall into the lower left part of the full grid.

6-11.3. Response Time and Response Time Differences

In both sets N and R, the response times grow as the proportion of CPU execution actions grow. The difference between the APART response time and the TOGETHER response time grows as CPU use grows; it is more important to select the faster configuration for large, long-running programs than for short ones. As CPU use increases, TOGETHER becomes slower much more quickly than APART.

In set N, the mean difference between the TOGETHER and APART response times is about 30% of the response time; however, the differences range from less than 10% in the upper left of the full grid to almost 50% on the right and the bottom. In set R, the mean difference is about 4.5%, and again there tend to be smaller differences in the upper left and larger ones on the right and the bottom.

We propose the always-APART placement policy, which is simply to always use the APART configuration; this policy would be simple to implement and fast to run. It would select the faster configuration in every run in set N except those in the extreme upper left cell. Even in set R, always-APART would select the faster configuration in about 77% of the runs. If APART were used when TOGETHER would be faster, the penalty would be the difference in response times. In set R, the mean penalty is 0.37% of response time, and the largest penalty is 4.50%. We believe that real programs will probably not fall into the zoomed-in grid at all, and, if they do, they will probably not fall into the upper left part of it (the *transition zone*) which is the only region in which TOGETHER is faster. Thus, for practical purposes, the always-APART policy is nearly perfect.

6-11.4. Statistical Analyses

We sought to identify the characteristics which would allow us to predict t_{diff} (the difference between the response time of the TOGETHER configuration and the APART configuration) and hence would allow us to predict which configuration would be faster. We used three groups of variables: (1) summary variables, drawn from the program traces, characterizing the program source and input data; (2) execution behaviour variables, drawn from the simulation outputs, characterizing the program, input data, and the distributed system; and (3) computation-to-communication ratios, or CCRs, estimates based on the program traces and some simplified assumptions about the system characteristics.

Using the correlation, stepwise regression, and regular regression procedures of the SAS statistics package, we identified a group of 17 variables to use in further analysis. We found a group of four

variables which contributed most of the predictive power of the 17 variables, which we further narrowed down to one variable, *smextime*, the mean per-process CPU service time requested in execute actions, which in turn contributed most of the predictive power of the four. For set N, the R^2 with 17 variables is 0.96, with four variables 0.91, and with one variable 0.86; for set R, the R^2 with 17 variables is 0.74, with four variables 0.66, and with one variable 0.65. The single variable *smextime* seems to capture most of the predictive power of the larger sets of variables.

In a digression from studying t_{diff} and seeking to select the faster configuration, we analyzed the amount of time processes spend waiting for messages to travel through the communication network (communication delay) and the time they spend waiting for other processes to send them messages (synchronization delay). We call the ratio of these the synchronization proportionality factor, or SPF. In set N the mean SPF is about 153, that is, processes spend about 153 times longer waiting for other processes to send them messages than they wait for the messages to arrive. Even in set R, which is extremely communication-intensive, the mean SPF is about 4. These SPF values are remarkably high, especially considering that our traces are generated so that processes are unlikely to do a receive unless another process has sent or soon will send a message to them. We conclude that the best way to improve the performance of distributed programs of the type we are studying is to improve the synchronization, or to avoid blocking receives.

6-11.5. PRED1 and PRED4

We constructed programs, PRED1 and PRED4, which compute t_{APART} (the response time of the APART configuration), t_{TOG} (the response time of the TOGETHER configuration), and t_{diff} (the difference between the TOGETHER and APART response times) according to the one-variable and four-variable formulas given by the SAS regression procedures, respectively. For t_{APART} and t_{TOG}, the predicted (computed) values are quite close to the actual values. For t_{diff}, the mean difference between the predicted value and the actual value is small, but the mean relative error is quite large (-101% for PRED1, -75% for PRED4). Furthermore, t_{diff} can be positive or negative; the mean relative magnitude of the error, that is, the mean relative error of $|t_{diff}|$, is very large (430% for PRED1, 362% for PRED4). Part of these large mean errors are due to eight or nine large errors in the thousands of percents and one huge error in the tens of thousands of percents, but even without the one huge error, the mean relative magnitude of the error is 252% for PRED1 and 216% for PRED4. The formulas for predicting t_{diff}, while good on average, are not very accurate in individual cases.

The sign of t_{diff} indicates which configuration is faster. Using the signs of the t_{diff} values computed by PRED1 and PRED4, we selected a configuration for each run in set R and compared it to the actual faster configuration. For both PRED1 and PRED4, the predictions were correct in about 81-82% of the runs. The two predictions differed in only three runs, all of which are in the 1% CPU row of the zoomed-in grid. Most of the incorrect predictions occur in the top two rows of the grid, and all of them occur in the top four rows, those with 3% or less CPU actions. We do not believe that any real distributed programs will fall into that part of the grid, so we can say that for practical purposes, PRED1 and PRED4 are perfect. Of course, below that part of the grid, APART is always best, so always-APART is also perfect, and much simpler to implement and faster to use.

6-11.6. Experiment Set S

In experiment set S, we generated traces for 12 cells in the zoomed-in grid without using the termSync feature of GENACTIONS. These traces average somewhat longer than the traces in sets N and R, and the mean difference between the number of actions for each of the two processes per program is larger than that for set R but smaller than that for set N. The winning configurations in set S are similar to those in set R, but the R^2 results for set S are smaller than for set R, smaller even than for the subset

of R consisting of the cells matching set S. If real programs are more like set S than set R, the success of the PRED1 and PRED4 techniques might be less than the 80% we achieved in set R.

6-11.7. Experiment Set T

In experiment set T, we generated traces that all have the same characteristics except for their length; the lengths vary from 100 to 1,000 actions per process. With 100 actions per trace, two of the four repetitions per case were APART wins; with 200 or more actions per trace, there were three or all four APART wins. The response time appears to grow approximately linearly with the trace length; although the errors go from about 2% to about 20%, there is no pattern in the sizes of the errors. The relative percentage differences between the response times of the two configurations are all between 1.4% and 2.2% for trace lengths from 400 to 1,000 actions per trace, for shorter trace lengths the relative percentage differences are larger. We conclude that trace lengths of 400 actions per process or more are sufficient, and indeed the traces in our other experiment sets all have approximately 400 actions per process.